UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

TRABAJO FIN DE GRADO

Evaluación de prestaciones de Entornos de Red Virtualizados

Alejandro Esparza Juandeaburre
Tutor: Jorge E. López de Vergara Méndez

JULIO 2016

Evaluación de prestaciones de Entornos de Red Virtualizados

AUTOR: Alejandro Esparza Juandeaburre
TUTOR: Jorge E. López de Vergara Méndez

High Performance Computing and Networking Research Group

Dpto. de Tecnología Electrónica y de las Comunicaciones

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Julio de 2016

Resumen

Vivimos en la época de la virtualización, por ello a la hora de desplegar una red en la vida real, es importante hacerse una idea inicial virtualizada, para ver cómo se comportará y si cumplirá los requerimientos mínimos de Calidad de Servicio (QoS, *Quality of Service*). Esta idea inicial puede ayudar a identificar errores en la red previos al despliegue real y corregirlos con suficiente antelación o, simplemente, ver como funcionaria un nuevo servicio sobre una red ya desplegada sin interferir en ella. Otro aspecto muy importante de las herramientas de emulación, es el uso académico que tienen, ya que permiten de una manera sencilla y muy económica, implementar gran cantidad de topologías y realizar pruebas sobre ellas.

Este trabajo de fin de grado consiste en la determinación de la herramienta de emulación de redes *open-source*, disponibles en Linux, más completa y con un consumo moderado de recursos del ordenador anfitrión. Para dirimir que herramienta de emulación es la que mejor rendimiento presenta, se han elaborado una serie de comparaciones que nos orientarán hasta poder determinar que herramienta es la más adecuada. Se analizarán el consumo de memoria, consumo de CPU, tiempos de despliegue y ancho de banda máximo.

Un factor a tener en cuenta a lo largo de este trabajo de fin de grado es que siempre se habla de emulación y no de simulación. Con simulaciones nos referimos a aquellos procesos dirigidos por eventos, que es una manera económica y rápida de hacer experimentos con protocolos.

También se tratara sobre los métodos de virtualización principales en los que se basan dichas herramientas de emulación y las plataformas de *routing* que soportan (tales como Quagga).

Palabras clave

Open-source, QoS, Linux, emulación, simulación, *benchmarks*, métodos de virtualización, plataformas de *routing*, consumo de memoria, consumo de CPU, tiempos de despliegue, ancho de banda.

Abstract

We live in the age of virtualization, so when people want to deploy a network in the real life, it is primordial to have an initial idea of what we are going to achieve, to know how to it will work and if it will satisfy the minimum requirements of Quality of Service (QoS). This first approach could help identifying errors in the network in advance, and make that the problem could be fixed with enough time, or simply it can help trying new services in a network which emulates a real one, without affecting the real one. Other important fact is the academic use of the emulation tools, because it allows deploying different scenarios very easily and without spending any money and make many experiments on them.

This Bachelor Thesis consist in the choice of an open-source emulation tool, available in Linux which has the best characteristics and performance. To get a conclusion of which one would be the best emulation tool, different benchmarks have been done, which will guide us through the final conclusion on which tool is the best. We will focus on the memory usage, CPU usage, time to deploy the scenario and the provided bandwidth.

An important fact is that in this Bachelor Thesis we focus on emulation instead of in simulation. With simulations we refer to those processes managed by events, which is an economical way and quick to make experiments with protocols.

We are going to study the principal virtualization methods that the emulation tools uses, and routing platforms such as Quagga.

Keywords

Open-source, QoS, Linux, emulation, simulation, benchmarks, virtualization methods, routing platforms, memory usage, CPU usage , deployment times, bandwidth.

Agradecimientos

Finalizar etapas es ley de vida, y la consecución de objetivos es la parte visible de un esfuerzo que no siempre se valora hasta tiempo después. Ahora, con un poco de perspectiva, me doy cuenta en donde me encuentro realmente, y es algo que hace cinco años era inimaginable. El haber logrado llegar hasta este punto ha sido propiciado por gran cantidad de factores, uno de los más importantes mi familia. Agradecer todo el apoyo de mis padres, Andrés y Maite, y por supuesto de mi hermano Adrián, ya que han sido una pieza fundamental para que este hoy escribiendo este Trabajo de Fin de Grado.

También el resto de mi familia, que ha contribuido a hacer más llevaderos estos años, tíos primos y abuelos. No puedo dejar de agradecer la paciencia, el apoyo y sobre todo la comprensión de aquellos amigos que siempre han estado ahí, Álvaro S., Álvaro D., Borja G. Víctor F., Víctor C., Gabriel M., Carlos M., Nacho C., Sergio H., Ana I., Raquel J., Virginia M. y otros tantos que no voy a nombrar con la intención de hacer finita la lista. Gracias.

Agradecer también las nuevas amistades generadas a lo largo de estos años de universidad, en los que ha habido mejores y peores momentos, pero siempre hemos logrado solventarlos. Gracias.

No me puedo olvidar de los profesores que me han aportado las herramientas necesarias para valerme por mi mismo en el mundo de las telecomunicaciones. Mención especial a mi tutor de este Trabajo de Fin de Grado, Jorge. Muchas gracias por la comprensión y la disposición que me has ofrecido a lo largo de este año.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

Glosario

API	Application Programming Interface / Interfaz de Programación de Aplicaciones
QoS	Quality of Service / Calidad de Servicio
WAN	Wide Area Network / Red de Área Extensa
RIP	Routing Information Protocol / Protocolo de Información de Rutas
OSPF	Open Shortest Path First / Primero la Ruta más Corta
SDN	Software Defined Network / Red Definida por Software
NFV	Network Function Virtualization / Virtualización de las Funciones de Red
KVM	Kernel-based Virtual Machine / Máquina Virtual Basada en el Núcleo
VNX	Virtual Networks over linuX / Redes Virtuales sobre Linux
UML	User-Mode Linux / Linux en Modo Usuario
LXC	LinuX Containers / Contenedores Linux
CORE	Common Open Research Emulator / Emulador Libre Común de Investigación
IP	Internet Protocol / Protocolo de Internet
XML	eXtensible Markup Language / Lenguage de Marcas Extensible
WA	Washington
CA1	California 1
CA2	California 2
UT	Utah
CO	Connecticut
NE	Nebraska
TX	Texas
IL	Illinois
MI	Michigan
GA	Georgia
PA	Pennsylvannia
DC	Washington DC
NY	New York
NJ	New Jersey

1 Introducción

1.1 Motivación

La motivación de este TFG surge a raíz de la cantidad de recursos tanto físicos como económicos que se emplean a la hora de desplegar o modificar una red de comunicaciones. Por ello, una aproximación inicial de lo que vamos a implantar en la vida real nos puede dar una visión de los inconvenientes que nos vamos a encontrar. Y sobre todo nos da margen suficiente para solventarlos. Es tan grande el alcance de este tema que es muy interesante la virtualización de redes previo al despliegue real de las mismas. Además, es un tema que está a la orden del día y está en continuo desarrollo y cambio.

Este TFG también viene motivado por el abanico de herramientas de emulación *open-source* que podemos encontrar para la implementación de redes virtuales. Un análisis previo de estas herramientas nos dará la suficiente perspectiva para poder realizar una serie de pruebas con el fin de poder determinar cuál de ellas es la que más se asemeja a la vida real, cuál nos da más libertades en el diseño, y en consecuencia, cuál es la más apropiada para utilizar a la hora realizar un análisis previo al despliegue.

El mundo de la virtualización es muy amplio, por lo que nos vamos a centrar únicamente en lo que nos atañe realmente que es la virtualización de redes de comunicación en internet.

Para comenzar es muy importante conocer la diferencia principal entre tres conceptos que, a simple vista, podrían parecer muy similares pero albergan diferencias importantes entre ellos. Estos conceptos son: simulación, las pruebas en vivo y emulación, como se muestra en [1].

Podemos entender como simulación, en el ámbito que nos incumbe, aquella como un proceso dirigido por eventos, que es una manera económica y rápida de hacer experimentos con protocolos. Verdaderamente, los simuladores de redes son esenciales para hacernos una idea previa del funcionamiento de los protocolos que vamos a utilizar en nuestra red. No obstante no pueden reemplazar la evaluación práctica de los protocolos ya que únicamente es una aproximación. Dicha aproximación, tiene a su favor que es muy eficiente y económica, tanto es así que es posible simular una hora lógica en unos pocos milisegundos reales. El único inconveniente es que se centra en el análisis de protocolos.

Las pruebas en vivo se fundamentan en evaluaciones sobre implementaciones de red reales. La manera de realizar las pruebas es usar tecnologías reales con el entorno de red subyacente. Este entorno de red puede ser el objetivo de estudio u otro muy similar. Es una manera de probar software distribuido usando hardware o software real ya implantado para la realizar pruebas, por ejemplo, de un nuevo servicio. Un inconveniente de este método es que a la hora de hacer pruebas con WAN (Wide Area Network, Red de Área Extensa), el coste económico es muy elevado. Un ejemplo claro sería aquella red que involucre redes de comunicación con satélites.

Por último, y de lo que trata este TFG, nos encontramos con las redes emuladas. La emulación de redes se encuentra entre medias de las pruebas en vivo, que, como hemos

visto anteriormente, permiten hacer gran cantidad de ensayos sobre hardware ya desplegado, y la simulación, que permite testear protocolos en entornos simulados. Esto quiere decir que la emulación nos permite crear escenarios con total libertad y sobre estos escenarios podemos ejecutar y medir rendimiento de protocolos reales e implementaciones de aplicaciones en tiempo real. El propósito de la emulación es crear entornos de comunicación controlados para realizar pruebas. Estos entornos emulados pueden perseguir objetivos específicos de QoS (*Quality of Service*, Calidad de Servicio), ya que la emulación de entornos de red nos da la libertad de emular equipos, cables, topologías...

En resumen la diferencia principal entre simulación, pruebas en vivo y emulación es, que las dos primeras se centran en aspectos concretos, por ejemplo, en simulación solo podremos hacer pruebas sobre protocolos. En pruebas en vivo podemos estudiar servicios sobre hardware ya desplegado, pero esto puede ser costoso o incluso imposible de verificar nuevos servicios debido a que el hardware instalado no soporte los requerimientos de la nueva tecnología a desplegar. Por último la emulación es una mezcla de los dos conceptos anteriores, lo que quiere decir que nos permite crear topologías al antojo del desarrollador para hacer pruebas concretas. Por ello este TFG va tratar sobre las diferentes herramientas *open-source* de emulación, actualmente disponible, para ver cuál es la que mejor rendimiento aporta, consumiendo el mínimo de recursos de un ordenador ya que es donde se van a emular las redes completas.

1.2 Objetivos

El objetivo principal de éste TFG es la determinación de una herramienta de emulación de redes *open-source* (de código abierto y libre distribución) para distribuciones Linux, la cual nos aporte la mayor fiabilidad y objetividad con respecto a una red que nos podríamos encontrar en la vida real. Es decir, que nos dé la libertad para alcanzar unos parámetros de QoS establecidos, siempre utilizando un uso razonado de los recursos del ordenador anfitrión donde se realicen las pruebas. La idea final es que haya un equilibrio entre uso de recursos junto con prestaciones de la herramienta de emulación. Con esto nos referimos, por ejemplo, a libertad para el diseño, introducción de impedimentos en la red, capacidad de incluir diferentes tipos de marcas que se utilizan en el mercado actualmente (*routers*, *switches*), etc.

En resumen los objetivos principales son:

- Investigación de las principales herramientas de emulación de redes *open-source* para entornos de Linux.

- Configuración de la topología que vamos a estudiar, para posteriormente poder realizar un análisis lo más justo posible.

- Realización de bancos de pruebas para comprobar los rendimientos de cada herramienta bajo análisis, partiendo de ideas tales como las planteadas en [2].

- Análisis de los resultados obtenidos y determinación de cuál es la herramienta de emulación que más se ajusta a la realidad manteniendo un uso discreto de los recursos del ordenador anfitrión.

1.3 Fases de realización

La evaluación de prestaciones conlleva un camino previo que comienza con entender cómo trabajan y en que se fundamentan las herramientas sobre las que se van a realizar las pruebas posteriores. Una vez que ya se tiene una visión del funcionamiento de cada herramienta de emulación se podría proceder al desarrollo de la topología deseada utilizando todas las facilidades que nos puede proporcionar cada herramienta.

Después de lograr tener el escenario creado hay que configurar un método de comunicación entre los nodos, ahí es cuando entra la valoración de los protocolos de *routing*. Donde también se ha dedicado tiempo en la elección del más apropiado. En las fases iniciales se sopesó la utilización del protocolo de routing RIP (*Routing Information Protcol*¸ Protocolo de Información de Rutas), pero al poco se descartó por la limitación que tiene de que, pese a que calcule el camino más corto para llegar al destino, tiene un límite de saltos, concretamente 15. En nuestro caso podría servir perfectamente, pero con la intención de que sea lo más genérico y que esté preparado para modificaciones, se escogió OSPF (*Open Shortest Path First*, Primero la Ruta más Corta), que no impone ninguna limitación de saltos.

Una vez que todos los nodos tenían conexión y se podía llegar a ellos por medio de "ping" podríamos decir que el escenario ha quedado completado. Pasaríamos al punto fuerte de este trabajo que es ver que pruebas realizar para poder determinar que herramienta sea la más favorable. Las pruebas que se van a hacer son de consumo de CPU, consumo de memoria, tiempo de despliegue y en último lugar de prestaciones de cada herramienta en cuanto a libertad del diseño.

En último lugar se han derivado las conclusiones en base a los resultados obtenidos en las pruebas. Las fases a modo de esquema serían las siguientes:

- Documentación: Durante los meses finales del curso 2014/2015, los meses de verano y primeros meses del curso 2015/2016. Para ello se buscó información sobre la emulación, simulación y sobre las herramientas *open-source*. Se leyeron artículos, tales como [3], [4] entre otros y trabajos anteriores [5].

- Diseño del escenario: Durante los meses previos al 2016 se comenzaron a implementar los escenarios, finalizándolos completamente sobre el mes de abril de 2016.

- Realización de pruebas: Según se iba finalizando un escenario se iban adquiriendo los datos necesarios para las distintas pruebas. La recopilación y organización completa de los datos se llevó a cabo desde la finalización de la implementación de los escenarios hasta el mes de mayo de 2016.

- Análisis de datos: Una vez que se tuvieron todos los datos se procedió a la comparación de todos ellos en conjunto. Hasta mediados de junio de 2016.

- Redacción de la memoria: Como uno de los pasos finales, se redactó la presente memoria del Trabajo de Fin de Grado, para plasmar los conocimientos y las conclusiones a lo largo de la realización del mismo.

Con el fin de resumir gráficamente el progreso realizado durante la elaboración de este Trabajo de Fin de Grado se incluye el siguiente Diagrama de Gant:

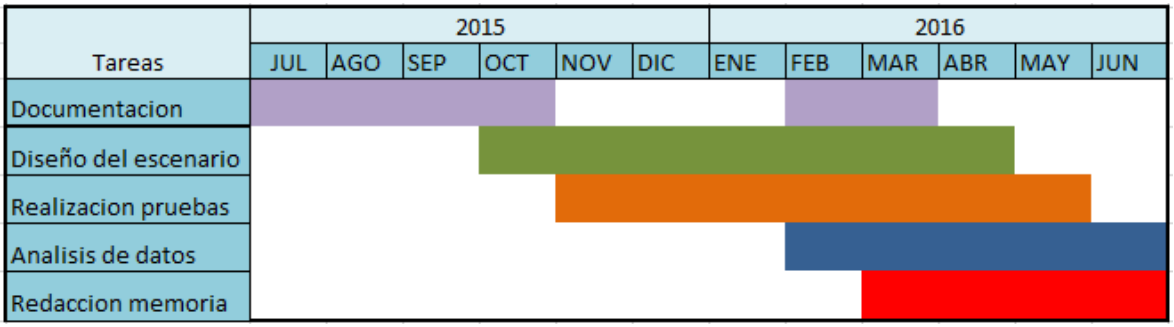

Figura 1-1: Cronograma de realización del Trabajo de Fin de Grado

1.4 Organización de la memoria

La memoria está estructurada en los siguientes capítulos:

- En el capítulo 2 se introduce, la situación en la que se encuentran las principales tecnologías que se utilizan para emulación de redes. También se detalla el punto de desarrollo en el que se encuentran las herramientas utilizadas a lo largo de este Trabajo de Fin de Grado y en qué se basan.

- En el capítulo 3 se muestra como se ha implementado el escenario bajo estudio en las diferentes herramientas.

- En el capítulo 4 se presentan las pruebas realizadas para poder desarrollar las comparaciones de los programas de emulación.

- En el capítulo 5 se aúnan todas las conclusiones obtenidas en el aparatado anterior para poder determinar, teniendo datos tangibles, cuál es la herramienta más completa y que menos recursos consume. En este capítulo también se ofrecen otras vías de proseguir este trabajo de fin de grado u otros trabajos futuros relacionados con el la emulación de redes.

2 Estado del arte

2.1 Introducción

A continuación vamos a repasar como se encuentra el estado del arte de los temas que abarcan este Trabajo de Fin de Grado. Pondremos nuestra atención en qué estado se encuentran las herramientas que se utilizan para virtualizar los entornos emulados que desplegaremos para el posterior análisis, es decir, qué métodos de virtualización se utiliza en las topologías que vamos a estudiar. Se destacarán los aspectos más característicos ya que no queremos extendernos en exceso.

En la figura 2-1 se muestra, a modo de resumen, el aspecto que tienen las máquinas virtuales. Donde estas incluyen las aplicaciones, los binarios necesarios y por supuesto el sistema operativo a emular. Lo cual ocupará en el ordenador anfitrión del orden de GB. Es recomendable tener en menta esta figura ya que después se hablara de virtualización con contenedores.

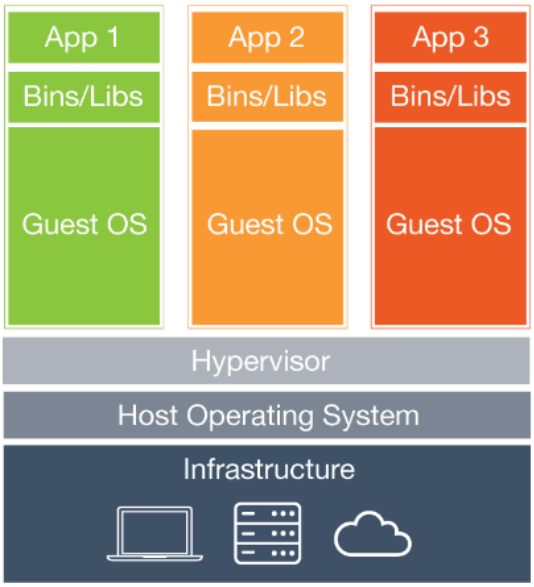

Figura 2-1: Máquina virtual convencional [10]

2.2 Herramientas de virtualización

En esta sección se van a tratar los distintos métodos de virtualización, sobre los cuales trabajan las herramientas de emulación bajo estudio, que se describen más adelante en la sección **¡Error! No se encuentra el origen de la referencia.**.

2.2.1 KVM

Kernel-based Virtual Machine (KVM) [6] es una infraestructura de virtualización para el núcleo de Linux. Los desarrolladores de KVM, en vez de crear la mayoría de partes de un sistema operativo, como otros *hypervisors* o hipervisores han hecho, han ideado un método que transforma el propio núcleo de Linux en un hipervisor. Para ello es necesario que el sistema anfitrión soporte virtualización de hardware (Intel-VT o AMD-V). KVM permite correr máquinas virtuales tanto de Linux como Windows. Cada máquina virtual tiene su

propio hardware virtualizado: tarjeta de red, disco duro, adaptador de gráficos etc. Hay que resaltar que KVM se desarrolla partiendo de los resultados previos de QEMU. La herramienta VNX permite utilizar este método de virtualización.

2.2.2 Dynamips

Dynamips [7] es un emulador de imágenes de *routers* que utilicen un procesador MIPS, tales como los *routers* de Cisco. Este método de virtualización utiliza una gran cantidad de memoria RAM y CPU, como se verá más adelante en esta memoria. Eso es debido a que dynamips usa archivos de memoria mapeada para la memoria virtual de los *routers*. El alto consumo viene por que emula la CPU del *router* instrucción por instrucción. Por ello hay una función llamada idle-pc que hace decrecer este alto consumo de CPU. Dynamips se apoya en dynagen, que es un front-end basado en texto. La herramienta GNS3 utiliza este método de virtualización, y también está soportada en VNX.

2.2.3 QEMU

QEMU [8] es un virtualizador y emulador *open-source*. QEMU permite ejecutar diferentes sistemas operativos por medio de la traducción dinámica, lo que le aporta un rendimiento bastante bueno. Actualmente se encuentra en la versión 2.6.0. QEMU soporta virtualización de sistemas x86 cuando se ejecuta usando KVM.

2.2.4 UML

User-Mode Linux [9] es una forma segura de ejecutar versiones y procesos del kernel de Linux. UML ofrece una máquina virtual que puede llegar a tener más recursos de hardware y software que la maquina sobre la que se ejecuta. Por ejemplo, se podrían añadir más interfaces de red. El almacenamiento en disco para la máquina virtual está contenido totalmente dentro de un archivo en el ordenador anfitrión, por lo que se puede acceder a él de manera sencilla y hacer modificaciones. VNX permite utilizar este tipo de virtualización.

2.2.5 Docker

Docker [10] difiere un poco de las virtualizaciones convencionales ya que está basado en contenedores, los cuales agrupan todo el software necesario en un sistema de archivos. Garantiza que siempre se ejecute de la misma manera, independientemente del entorno en el que este. Es una virtualización muy ligera, permite la mayoría de distribuciones Linux y Windows y es bastante segura.

Como se puede ver la figura 2-2 presenta diferencias con respecto a las máquinas virtuales convencionales (figura 2-1), ya que comparte el núcleo con otros contenedores en lugar de tener uno para cada máquina virtual generada. Imunes está basado en este tipo de virtualización.

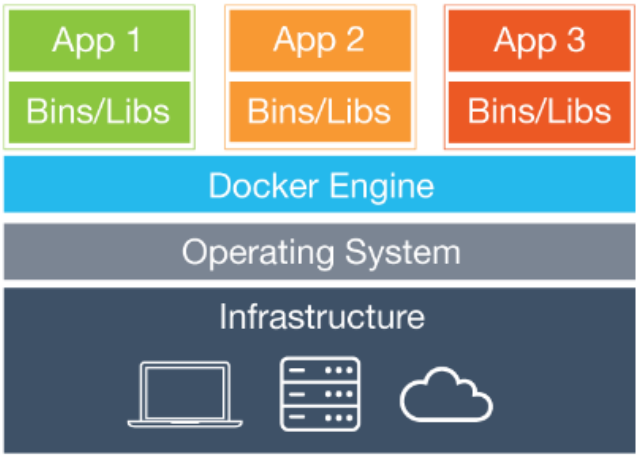

Figura 2-2: Contenedores Docker [10]

2.2.6 LXC

LXC (*Linux Containers*, Contenedores de Linux) [11] es una tecnología de virtualización exclusiva de Linux. Realmente no provee una máquina virtual como tal, sino que aporta un entorno virtual el cual cuenta con su propio espacio de procesos y redes. La idea principal es crear un entorno lo más similar posible a Linux pero sin la necesidad de tener un núcleo distinto. Es un método de virtualización ligero. VNX y CORE, entre otros utilizan este método de virtualización.

2.3 Herramientas de emulación open-source

Entre las herramientas de emulación *open-source* para entornos Linux, encontramos gran variedad. A continuación vamos a destacar las principales resaltando los factores que las caracterizan.

2.3.1 GNS3

GNS3 es una herramienta de emulación cuyo lanzamiento fue en el 2008. La versión con la cual se ha hecho las pruebas en este Trabajo de Fin de Grado es la versión 1.4.6. GNS3 es un emulador que consta de una interfaz gráfica que facilita mucho la implementación de los escenarios y ejecutar las simulaciones. GNS3 se basa para sus simulaciones en Dynamips (presentado en el apartado 2.2.2). La característica principal de esta herramienta es que ejecuta binarios de un fabricante en concreto CISCO. Ofrece la posibilidad de, con QEMU (presentado en el apartado 2.2.3) y virtualbox, simular cortafuegos. Igualmente ofrece VPCS, un emulador de PCs con funciones de red [12].

2.3.2 CORE

CORE *(Common Open Research Emulator*, Emulador Libre Común de Investigación), es un emulador de redes *open-source* que surgió como una derivación de Imunes para poder apoyar un proyecto de investigación de datos móviles. CORE utiliza como método de virtualización LXC (presentado en el apartado 2.2.6). Para la creación de escenarios consta de una interfaz gráfica muy intuitiva y sencilla de utilizar. Para el estudio de esta herramienta se ha utilizado la versión 4.8. Permite la utilización de gran variedad de elementos de red y protocolos de *routing* [13].

2.3.3 IMUNES

Imunes [14] está basado en el núcleo de Linux particionado en múltiples nodos virtuales, que pueden interconectarse con enlaces para crear todo tipo de topologías. Para establecer los escenarios consta de una interfaz gráfica lo que facilita mucho el proceso. Utiliza actualmente Docker como método de virtualización. Se ha utilizado la versión 2.1.0 de Imunes. Permite el uso de un amplio abanico de protocolos de encaminamiento de manera muy sencilla.

2.3.4 MARIONNET

Marionnet [15] surgió en la Universidad de Paris para facilitar la enseñanza de redes. Es una herramienta de simulación de alto nivel que permite reproducir con gran fiabilidad el comportamiento de redes formadas por ordenadores, *routers*, *switches*, *hubs* etc. Los dispositivos pueden ser creados dinámicamente, eliminados o modificados, mientras el resto del experimento sigue en funcionamiento. Marionnet cuenta con una serie de imágenes propias para la virtualización. Utiliza UML como método de virtualización. Permite la posibilidad de probar protocolos de encaminamiento así como distintos servicios. Se ha utilizado la versión 0.90.6.

2.3.5 NETKIT-NG

Nekit-ng [16] es una herramienta de emulación que permite crear diferente tipos de topologías de manera sencilla. A lo largo del TFG se le denominara únicamente como Netkit. Ofrece la posibilidad de crear gran cantidad de dispositivos virtuales de red, *routers*, *switches*, *hosts* etc. Este programa, a diferencia de los anteriores, no cuenta con una interfaz gráfica, por lo que la implementación del escenario conlleva algo más de elaboración ya que hay que hacer unos archivos de configuración. Para la emulación de las maquinas usa UML. La versión que se ha utilizado para hacer los experimentos ha sido 3.0.4.

2.3.6 VNX

Virtual Network Over Linux (VNX) [17] es una herramienta de emulación desarrollada en la Universidad Politécnica de Madrid, a partir de la experiencia previa en VNUML (Virtual Network with User Mode Linux). Tampoco cuenta con una interfaz gráfica para diseñar los escenarios, si no que para ello se basa en archivos de configuración en XML. Entre sus características principales presenta la posibilidad de crear varias máquinas virtuales con distintos sistemas operativos. Soporta también distintos métodos de emulación. La versión utilizada para las pruebas es 2.0b.5839.

2.3.7 MININET

Mininet [18] es una herramienta capaz de crear redes virtuales de manera muy realista. Mininet está enfocada sobre todo para emular redes SDN (*Software Defined Networks*, Redes Definidas por Sofware). Con SDN nos referimos a aquellas redes en el cual quitan el control de la red al hardware y se lo dan a una aplicación de software denominada controlador. Mininet, fue desarrollada por un conjunto de profesores de la Universidad de Stanford como herramienta para la investigación y para la enseñanza. Mininet permite crear escenarios que dependan de controladores OpenFlow. Para crear las topologías se utiliza la API de Python de Mininet, la cual es algo densa para usuarios inexpertos en Python. Para crear las máquinas virtuales utiliza los denominados "*network namespaces*", similares a LXC, los cuales proveen una manera muy ligera y sencilla de crear nodos

virtuales, pero no ofrecen las mismas funcionalidades que las máquinas virtuales convencionales. Se ha utilizado la versión 2.3.0d1.

2.4 Conclusiones

En este apartado se han presentado brevemente algunas características sobre las herramientas que van a analizarse y sobre que están fundamentadas. Es importante tener en mente los diferentes métodos de emulación que utiliza cada herramienta, esto nos ayudará a obtener conclusiones y comprender el motivo.

En el próximo apartado se detallara como se ha implementado la topología bajo análisis en cada una de las herramientas de emulación seleccionadas.

3 Diseño y desarrollo

3.1 Introducción

A lo largo de este capítulo vamos a detallar, de una manera general, como se desarrollan e implementan los escenarios en las diferentes herramientas bajo estudio, presentadas en el capítulo anterior.

En todas las herramientas se ha diseñado la misma topología. La topología escogida ha sido, la comúnmente conocida, NSFNet [19], que en sus orígenes fue la red destinada a la comunicación de la investigación y de la educación en Estados Unidos.

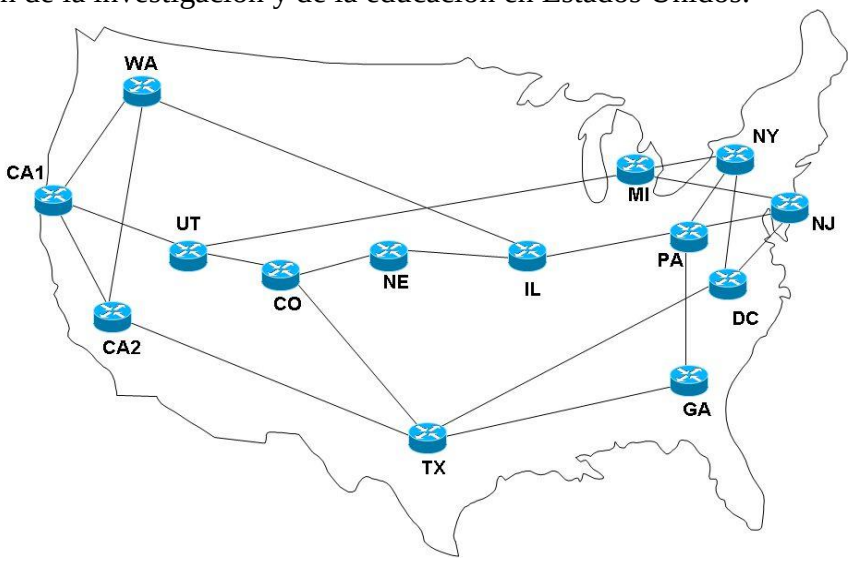

Figura 3-1: Topología NSFNet

Para que los diseños sean lo más justos posibles a la hora de realizar las comparaciones se ha utilizado para todos OSPF como protocolo de *routing*, salvo en mininet que por limitaciones del sistema anfitrión se ha tenido que desarrollar con rutas estáticas. El escenario está formado por *routers* y enlaces como se muestra en la figura 3-1.

3.2 Diseño del escenario

3.2.1 GNS3

Como hemos visto antes, GNS3 es una herramienta que nos permite emular escenarios con imágenes de *routers* de Cisco. Para crear el escenario en GNS3, en primer lugar tenemos que descargarnos las imágenes del *router* que queramos añadir a nuestra topología. Normalmente esta imagen se corresponderá con el router que tengamos disponible para implantar posteriormente, en caso de que se vaya a hacer un despliegue real. De esta manera podremos simular su comportamiento con mayor fidelidad y comprobar si se ajusta a las necesidades del usuario.

Una vez que tenemos las imágenes cargadas en GNS3 podemos añadirlas a la hoja de trabajo para ir creando la topología. Para nuestro estudio hemos escogido el *router* c3745. A continuación en la figura 3-2 se muestra la topología en GNS3.

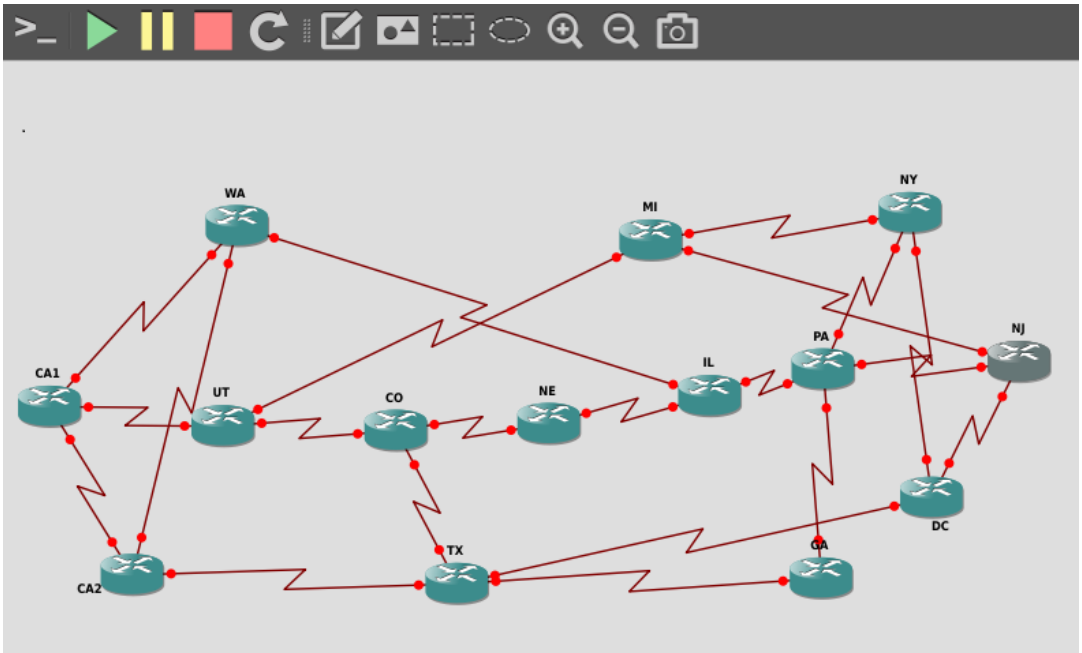

Figura 3-2: Topología de NSFNet en GNS3

Una vez colocados todos los *routers* con sus correspondientes enlaces, pasamos a configurar cada *router* de manera individual. A cada *router* se le ha asignado la misma memoria RAM (128MiB) e igual memoria NVRAM (*Non-volatile Random Access Memory*, Memoria de Acceso Aleatorio no Volátil) que es un tipo de memoria que no pierde la información que tiene almacenada al cortar la corriente eléctrica (se ha asignado 128KiB).

Después procedemos a la configuración de cada *router*, asignando las IPs correspondientes a cada interfaz y activando el protocolo de *routing* OSPF para que haya comunicación entre todos los nodos. A continuación se muestra la configuración de uno de los nodos, que se hace mediante CLI (*Command Line Interface*, Interfaz de Línea de Comandos):

Ejemplo de configuración de nodo CA1

```
>enable
#configure terminal

#interface s1/0
#ip address 10.0.0.2 255.255.255.0
#no shutdown
#exit

#interface s1/1
#ip address 10.0.2.1 255.255.255.0
#no shutdown
#exit

#interface s1/2
#ip address 10.0.3.1 255.255.255.0
```

```
#no shutdown
#exit

#router ospf 1
#network 10.0.0.0 255.255.255.0 area 0
#exit

#exit

#wr
```

Repitiendo este proceso, asignando las direcciones IPs correspondientes a cada interfaz, quedaría el escenario implementado. Posteriormente, para arrancar el escenario se podría levantar nodo a nodo si queremos hacer pruebas en secciones de la red o toda la topología a la vez.

3.2.2 CORE

CORE es un herramienta de emulación con interfaz gráfica, algo más sencilla que la comentada en el apartado anterior ya que no hay que cargar imágenes específicas de ningún *router*. Únicamente hay que arrastrar los elementos de la red que vayamos a utilizar (en nuestro caso *routers*) y establecer los enlaces oportunos para crear la topología deseada. Esta herramienta asigna automáticamente direcciones IP a cada interfaz al establecer los enlaces entre nodos, lo que es un aspecto a tener en cuenta. A continuación, en la figura 3-3 se muestra la implementación del escenario:

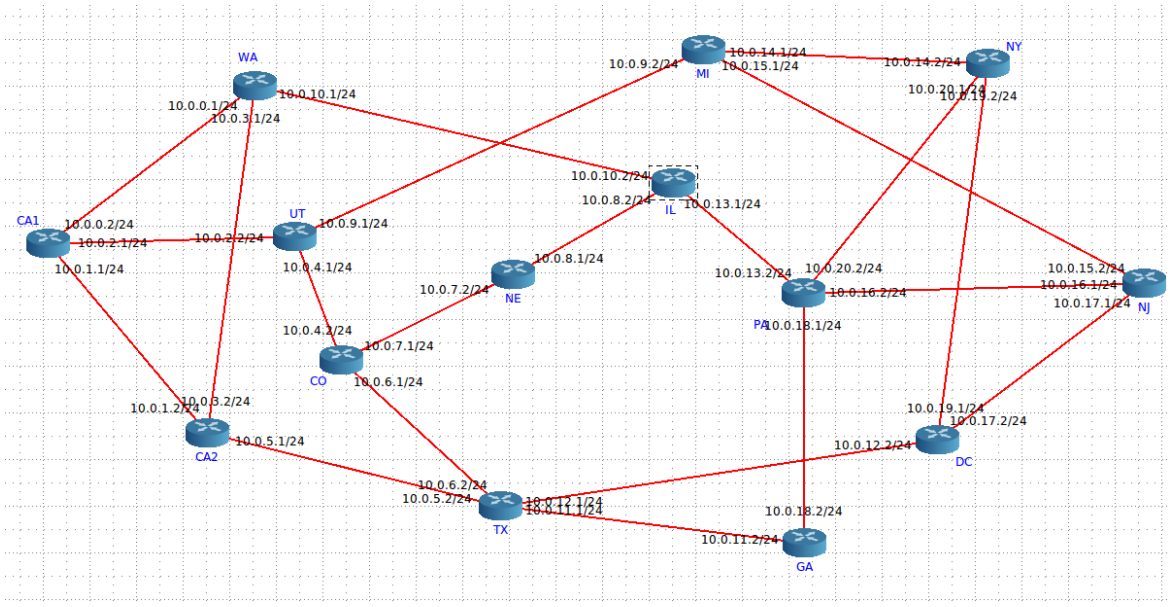

Figura 3-3: Topología de NSFNet en CORE

Una vez que ya tenemos todos los nodos configurados y conectados según la topología bajo estudio, esta herramienta, nos ofrece la posibilidad de ir activando en cualquier nodo el protocolo de *routing* deseado, en caso de que algún nodo utilice algún protocolo distinto. Si no, también está lo posibilidad de activar el mismo protocolo para todos los *routers* de una vez. Será lo que utilicemos para nuestro estudio. Hay que tener en cuenta que una vez que se arranca el escenario, el protocolo de *routing* (OSPF en nuestro caso), tarda unos

segundos en iniciarse ya que se tienen que completar las tablas de adyacencia de los *routers*. En la figura 3-4 se muestra un *router* con la tabla de encaminamiento completa:

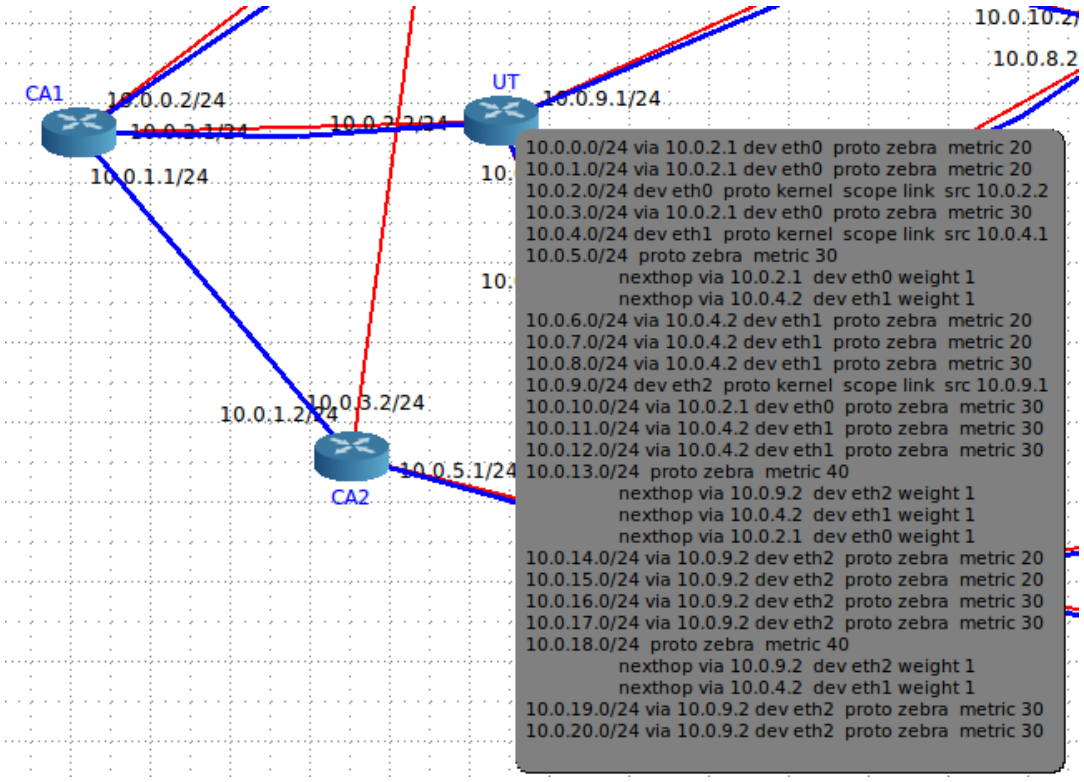

Figura 3-4: Con las rutas actualizadas de OSPF

Una vez que ya se han actualizado todas las rutas podemos comprobar si hay conectividad realizando ping entre nodos. Cada *router* tiene una terminal asociada en bash.

3.2.3 IMUNES

IMUNES es una herramienta de emulación con interfaz gráfica, lo que le aporta un punto a favor, o en contra si se diese el caso que tuviéramos que ejecutarlo en servidor en modo texto. Está basada en FreeBSD y en el kernel de Linux dividido en múltiples nodos virtuales ligeros, que pueden ser interconectados a través de enlaces para formar topologías de red arbitrariamente complejas, dependiendo del caso bajo estudio. La topología ya implementada quedaría de la siguiente manera (Figura 3-5):

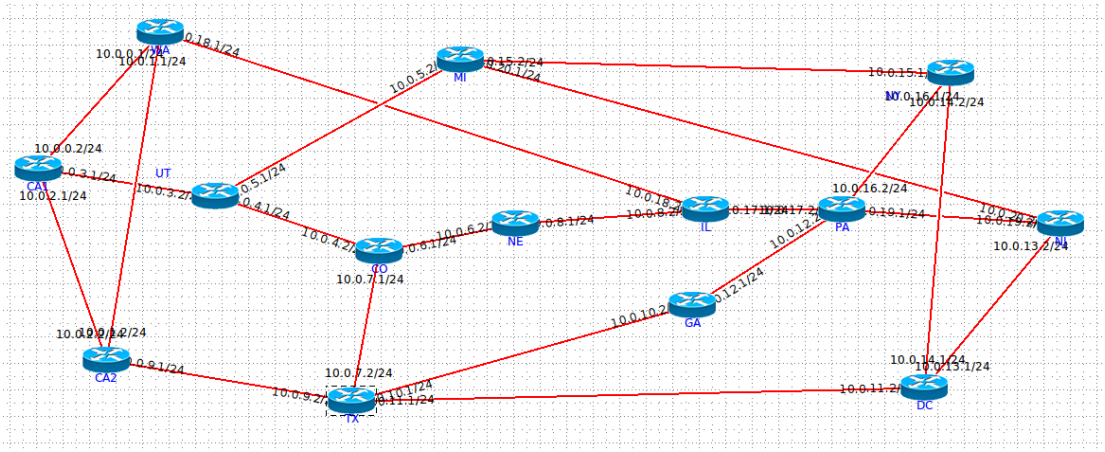

Figura 3-5: Topología de NSFNet en Imunes

Para crear dicha topología tenemos que arrastrar los *routers* de la barra lateral e ir conformando la topología, después establecemos los enlaces oportunos. Una ventaja de esta herramienta es que asigna automáticamente IPs a los nodos (cabe la posibilidad de cambiarlas a nuestro gusto). Una vez que tengamos todos los *routers* correctamente conectados podemos pasar a configurar el protocolo de *routing*. En esta herramienta es muy sencillo activarlo ya que únicamente tenemos que dar clic derecho en cada nodo, meternos en configuración y seleccionar el protocolo que deseemos utilizar (Figura3-6). En nuestro caso se ha utilizado Quagga (ver anexo A) [20] con OSPFv2 al igual que en el resto de programas.

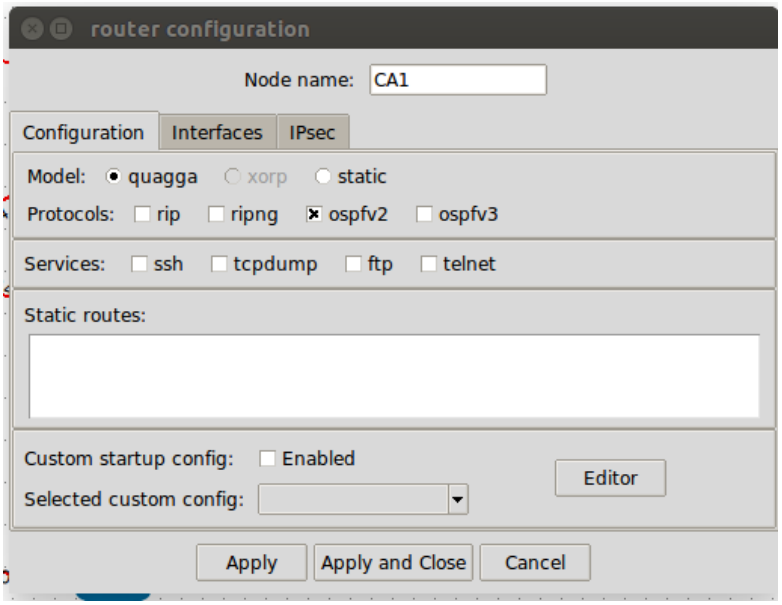

Figura 3-6: Configuración *router* en Imunes

En el momento que tenemos todo configurado correctamente podemos arrancar el experimento mediante `execute`. Este comando arranca todos los nodos, establece todos los links y activa los servicios de todos los *routers*. Una vez que termina, podemos comprobar la conectividad entre nodos con un `ping`.

3.2.4 MARIONNET

Marionnet al igual que el resto de herramientas analizadas hasta el momento, nos permite configurar e iniciar los equipos de una red sin la necesidad de medios físicos. Es una herramienta que está basada en una interfaz gráfica al igual que las anteriores. Por lo tanto cuenta a su favor con que se puede visualizar fácilmente cómo va quedando la topología.

Para crear la topología hay que ir añadiendo los nodos y los enlaces al proyecto. Una vez que esta todo en su sitio y correctamente conectado pasamos a configurar las direcciones IP manualmente ya que este programa no las auto asigna. (Aunque si se desease hay una opción que las asigna automáticamente las direcciones IP a los nodos. En nuestro caso las hemos asignado manualmente para que siga el criterio utilizado en los escenarios anteriores). La topología quedaría de la siguiente manera (Figura 3-7):

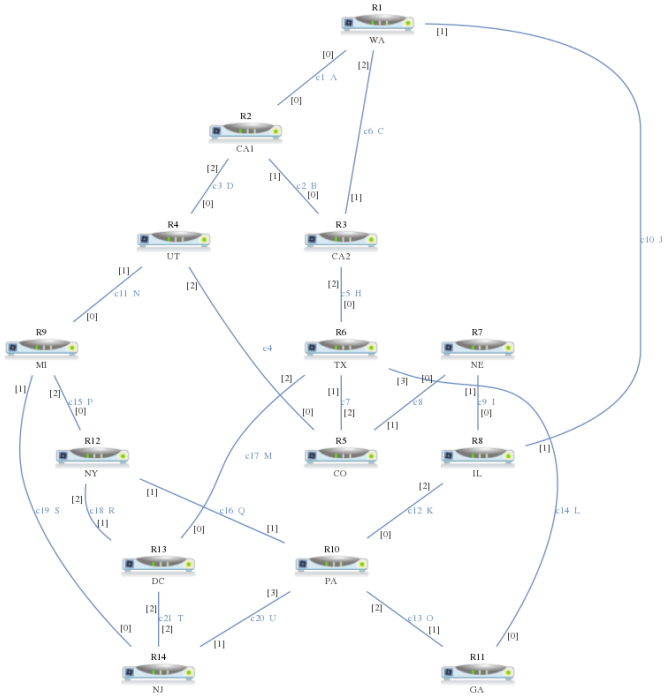

Figura 3-7: Topología de NSFNet en Marionnet

Una vez creada toda la topología procedemos a configurar el protocolo de *routing*. Para este caso es un poco más tedioso que en otras herramientas ya que para activar la tecnología de *routing* tienes que activarlo dentro de cada *router* por terminal. Una vez que estén todos ya configurados al arrancar el escenario se van actualizando las tablas y a los pocos segundos podemos hacer un ping para comprobar la conectividad. Con esto la topología quedaría diseñada y lista para medir el rendimiento.

3.2.5 NETKIT

El siguiente programa que vamos a utilizar para realizar el experimento es Netkit. Netkit presenta una forma totalmente distinta de implementar la topología a las herramientas anteriormente vistas. Esta herramienta de emulación, a diferencia de las demás, no está basada en una interfaz gráfica, lo que es una de las características que más lo diferencian del resto. Para crear la topología deseada lo que hay que hacer es un archivo de configuración en el cual aparecerán todos los *routers*, con sus correspondientes interfaces, y también como se interconectan todos los *routers*. Después pasaremos a implementar un

script para cada nodo de la red, en el cual definiremos las direcciones IP correspondiente a cada interfaz e iniciaremos el protocolo de *routing*. Para iniciarlo, tendremos que haber creado unos directorios en los cuales se encontrará Quagga que es el demonio que utilizamos para poder utilizar el protocolo de *routing* OSPF. Para poder ver la topología en esta herramienta podemos utilizar el comando `linfo -m [name]`. Al hacerlo en nuestra topología nos sale lo siguiente (Figura 3-8):

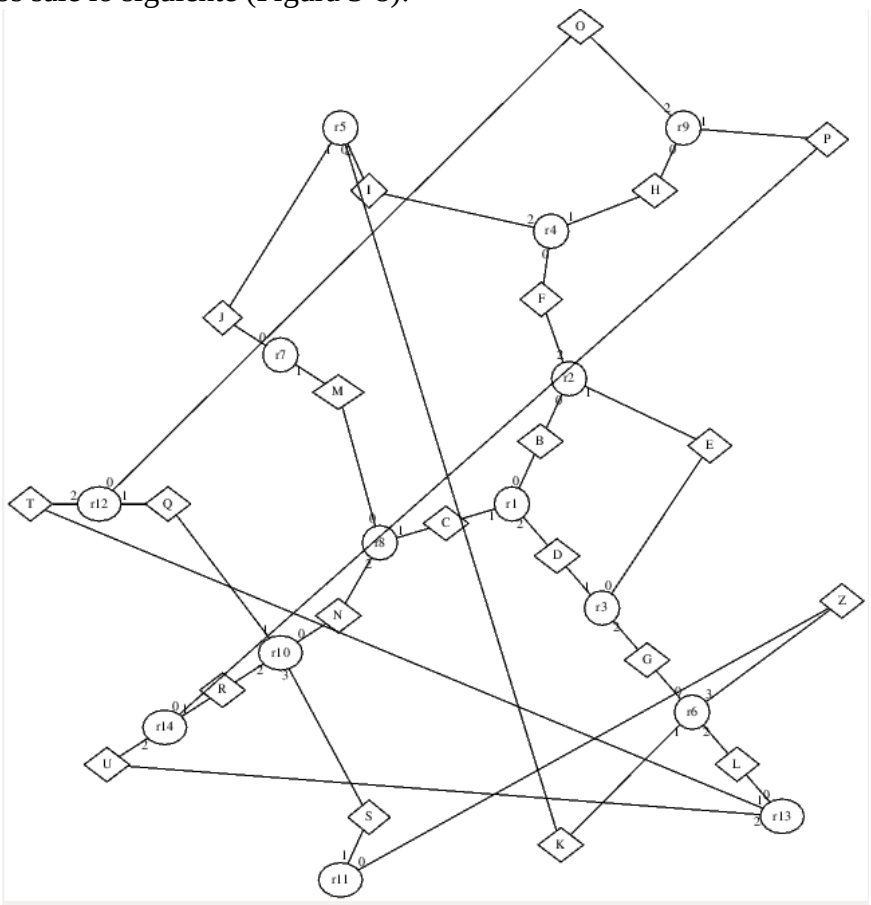

Figura 3-8: Topología de NSFNet en Netkit

Una vez que ya tenemos todo configurado y arrancado para asegurarnos que todo funciona correctamente hacemos `ping` entre todos los nodos. Una vez que llegamos a todos los nodos con `ping`, podemos afirmar que el escenario ha quedado correctamente implementado.

3.2.6 VNX

La herramienta de emulación VNX está basada principalmente en dos partes, una parte de lenguaje XML el que permite describir el escenario virtual y la segunda parte del programa VNX que parsea el XML y crea el escenario virtual sobre una maquina Linux. Por lo tanto para crear este escenario hay que tener alguna noción sobre XML ya que la descripción de los escenarios está enteramente especificada en dicho lenguaje (ver Anexo B).

Para generar el escenario lo que tenemos que hacer es establecer los enlaces entre los *routers*, posteriormente crear cada *router* (creados con LXC "linux containers", al ser más ligeros que las otras opciones: UML y KVM) asignándole sus direcciones IP correspondientes. La topología quedaría de la siguiente manera (Figura 3-9):

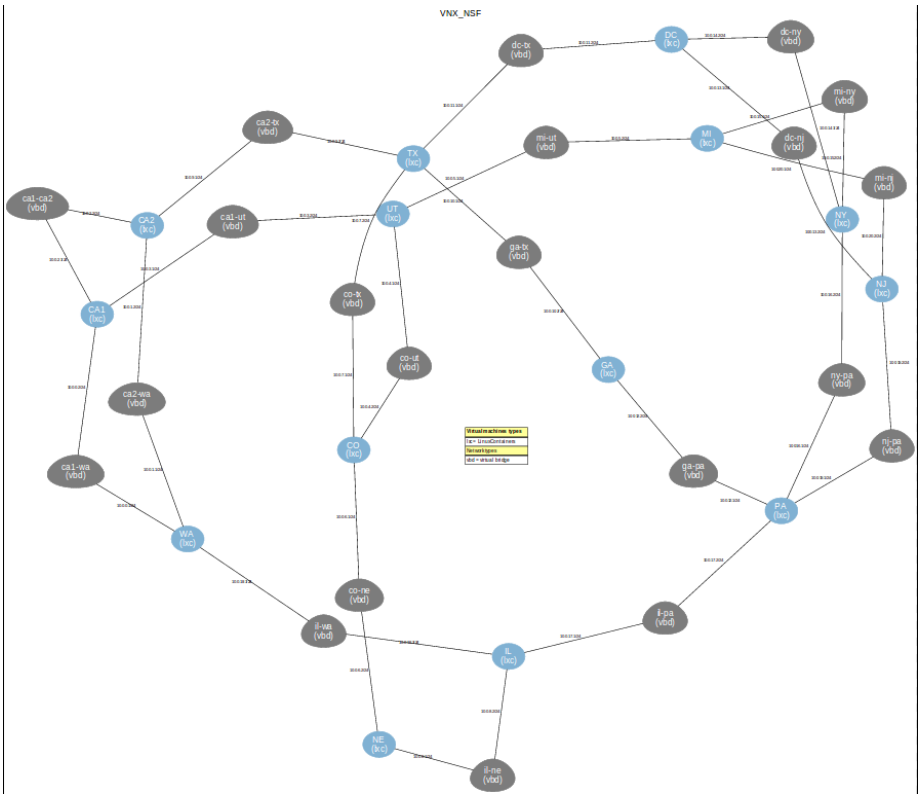

Figura 3-9: Topología de NSFNet en VNX

Cuando ya tenemos toda la topología configurada, solo nos faltaría activar el protocolo de *routing* para que haya conectividad entre todos los nodos. En este programa necesitamos un *plugin* para poder activar los servicios de OSPF. La configuración de dicho *plugin* también se describe en XML. Hay que tener en cuenta que tarda varios segundos en inicializarse mientras rellena las tablas de adyacencia entre *routers*.

Una vez que ya se han actualizado todas las rutas podemos comprobar si hay conectividad realizando `ping` entre nodos. Cada *router* tiene una terminal asociada.

3.2.7 MININET

Mininet es una herramienta de virtualización capaz de crear una red virtual, sobre un kernel real en un sistema anfitrión en escasos segundos.

La creación de la topología en esta herramienta de emulación está basada en Python(ver nexo B). Por lo tanto, para la implementación del mismo se ha creado un script en Python en el cual se definían todos los nodos de la red, con sus correspondientes configuraciones, así como también los enlaces entre ellos.

Los desarrolladores recomiendan instalar Mininet en una máquina virtual ya que puede afectar al sistema anfitrión. En nuestro caso lo hemos instalado en el anfitrión para que hubiese igualdad de condiciones a la hora de comparar los resultados de las pruebas. En la figura 3-10 se muestra la topología de NSFNet en Mininet:

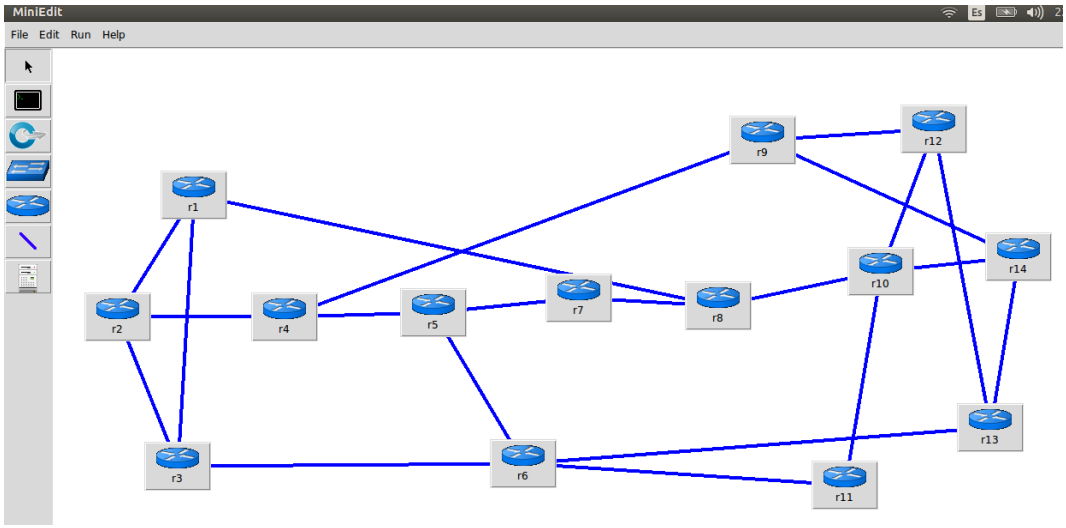

Figura 3-10: Topología de NSFNet en Mininet

Cabe destacar que también posee una interfaz gráfica para poder crear los escenarios, lo que simplifica en gran medida el proceso de desarrollo del escenario.

Para que haya conectividad en estos nodos se han establecido rutas estáticas, en vez del protocolo de *routing* utilizado en todas las demás herramientas. Este es un factor a tener en cuenta y le restará puntos a la hora de decidir qué herramienta es la que más prestaciones nos aporta con el compromiso de no consumir gran cantidad de recursos del equipo anfitrión.

Una vez que tenemos las rutas definidas comprobamos con `ping` que hay conectividad entre los nodos y así quedaría la topología plenamente implementada.

3.3 Conclusiones

En este apartado se ha resumido la manera de poder implementar un escenario en las distintas herramientas de emulación que vamos a analizar. Se ha visto como cada una de ellas tiene mayor o menor complejidad, como por ejemplo, algunas herramientas de emulación nos permitían escoger directamente el nodo de la red y colocar sobre nuestra topología por medio de una interfaz gráfica, otras en su lugar requerían de más esfuerzo y tiempo para su implementación. Esencialmente las que más tiempo han llevado son aquellas que basaban su implementación en algún lenguaje de programación, como bien puede ser Python, XML...

Un aspecto muy importante, es tener soltura para poder implementar diversos escenarios incluyendo amplias variedades de elementos que nos podríamos encontrar en una red real como bien pueden ser *hosts*, *routers*, *switches*, *bridges* etc.

También se ha definido como añadir protocolos de *routing* a las herramientas, concretamente OSPF, entre otros.

4 Integración, pruebas y resultados

4.1 Introducción

Una vez que ya tenemos una visión general de cómo se implementa el escenario que vamos a analizar en cada herramienta, pasamos a realizar un análisis más exhaustivo de todo el partido que se puede sacar a las distintas herramientas de emulación, comparándolas entre sí por medio de una serie de bancos de pruebas. Estos bancos de pruebas nos darán evidencia de que programas son los que tardan menos en iniciarse, cual consume menos recursos de CPU y de memoria del sistema anfitrión. También analizaremos los aspectos más importantes de cada herramienta, con esto nos referimos a la cantidad de libertad en el diseño que nos ofrece. Como bien puede ser la elección de los *routers* (distintos fabricantes, modelos...), introducción de retardos o pérdidas en la red que es un factor importante a tener en cuenta, manipulación de anchos de banda entre otros. Otro aspecto es la facilidad para crear el escenario sin tener conocimientos avanzados de programación; es decir, la sencillez de implementación del escenario dependiendo de la herramienta.

En resumen, el análisis nos permitirá determinar cuál es la herramienta que mejor se ajusta a una aplicación real, cumpliendo un compromiso de utilización de recursos moderados del sistema anfitrión y aportándonos un amplio abanico de posibilidades a la hora de implementar distintos escenarios.

Todas las pruebas se han realizado sobre el mismo equipo anfitrión, en distintas situaciones. Estas situaciones las hemos denominado de dos maneras; una primera llamada "medidas en frío", que involucra aquellas medidas que se han realizado recién encendido el equipo anfitrión sin tener ningún otro programa interfiriendo las medidas. Posteriormente la segunda manera llamada "medidas en caliente", se refiere a aquellas medidas que han sido tomadas después de haber iniciado el mismo escenario previamente y de igual manera se analiza de manera aislada la herramienta de emulación, es decir, sin ninguna otra herramienta ejecutándose en paralelo.

Cabe destacar que todas las pruebas se han desarrollado y medido con un equipo que presenta las siguientes características:

ACER ASPIRE 5750G de 64 bits que como procesador tiene un Intel® Core™ i7-2630QM CPU @ 2.00GHz (8 CPUs), a 2.0 GHz. Memoria RAM de 4096 MB y 500 GB de HDD. Para el desarrollo de todas las pruebas se ha trabajado en el entorno Linux con la versión de Ubuntu 14.04 LTS. Para ello se creó una partición del disco duro con 100GB.

Otro aspecto a tener en cuenta es, que los escenarios se han creado lo más similares posibles para que las medidas que tomemos sean comparables. De esta manera todos los escenarios presentan igualdad de condiciones, siempre teniendo en cuenta la libertad o restricciones que nos puede ofrecer cada herramienta.

4.2 Tiempos de despliegue

La primera prueba que vamos a realizar es aquella que involucra los tiempos de despliegue y tiempos de ejecución de cada herramienta de emulación. Hemos definido esta prueba como el tiempo que transcurre desde que arrancamos el escenario hasta el punto en el que

tenemos conectividad entre los nodos de la red emulada. Es decir también hemos incluido el tiempo que tarda en iniciarse el protocolo de *routing*.

Para la obtención de los tiempos hemos utilizado el comando `time` que está disponible en Linux, el cual, a modo de resumen, consiste en medir el tiempo de ejecución de un programa.

Algunas herramientas de emulación reportaban directamente el tiempo que habían tardado en ejecutarse. A la hora de tener en cuenta la inicialización del protocolo de *routing*, se comprobó cómo en algunas herramientas se iniciaba durante el proceso de creación de las virtualizaciones pero en otros había que activarlo posteriormente. Este tiempo se ha tenido en cuenta y se ha sumado al tiempo de arranque.

A continuación se muestra la figura 4-1, donde aparecen los tiempos de despliegue de las 7 herramientas utilizadas para el análisis. Se presentan las dos situaciones en las que vamos a comparar los resultados, en frío (columnas azules) y en caliente (columnas naranjas).

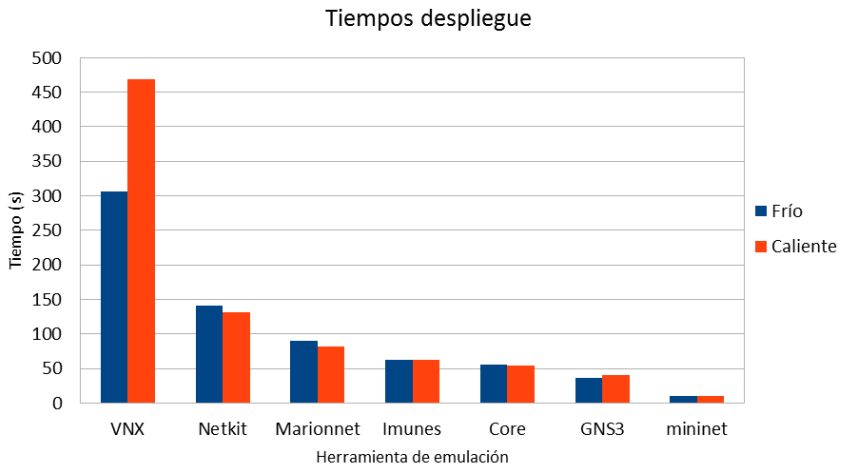

Figura 4-1: Gráfica tiempos de despliegue

Observando estos resultados podemos apreciar cómo sin duda alguna el programa que más tiempo tarda en iniciarse es VNX, y no solo eso, sino que a diferencia del resto, la medida en caliente no mejora el resultado obtenido, de hecho empeora bastante. Se podría pensar que es un hecho puntual pero de todas las herramientas se tomaron 7 medidas tanto en frío como en caliente, y se realizó una media aritmética que es el valor que se muestra finalmente en la gráfica. De esta manera podemos descartar que es un hecho puntual, y es algo que pasa todas las veces que se arranca el escenario por segunda vez.

También que tarde mucho más en arrancarse en caliente, nos da la idea de que no queda nada cacheado en el sistema una vez que se ha finalizado el escenario, como podría pasar perfectamente en el resto de herramientas que se puede apreciar como tardan algo menos en arrancar o el mismo tiempo, pero nunca más, salvo GNS3.

El siguiente programa que requiere un mayor tiempo de arranque es Netkit que tarda un poco más de dos minutos en arrancar la topología y los servicios. Es un tiempo aceptable aunque comparando con el resto de herramientas es bastante elevado.

Netkit virtualiza sobre UML (User-mode Linux) (ver apartado 2.2.4). Se puede apreciar que Netkit en caliente reduce un poco el tiempo de arranque, no es una mejora muy grande pero nos dice que sí que se queda cacheado algo en el sistema.

El siguiente programa que encontramos, que ya está por debajo de los dos minutos, es Marionnet. Esta herramienta ya presenta una mejora en tiempo de despliegue considerable con respecto a los dos programas comentados en líneas anteriores.

Marionnet también, al igual que Netkit reduce su tiempo de despliegue cuando se ejecuta en caliente. Ronda el minuto y medio, que teniendo en cuenta que hay que arrancar 14 nodos y los servicios de la red, está bastante bien.

Ahora vamos a hablar de los programas que mejor resultado obtienen en este apartado, ya que sus tiempos de ejecución y despliegue se encuentra alrededor del minuto o inferior.

Los primeros que encontramos son Imunes y CORE, que se van a analizar juntos debido a sus grandes similitudes ya que parten de un punto común. Tanto es así que CORE surgió a raíz de Imunes como hemos visto anteriormente. Core utiliza LXC como método de virtualización, mientras que Imunes utiliza Docker. Los tiempos son algo menores en CORE y se puede apreciar una pequeña mejoría al tomar las medidas en caliente. Imunes por su parte permanece inmutable y conserva más o menos el mismo tiempo de despliegue en ambas medidas.

A continuación GNS3 se encuentra por debajo de los 50 segundos, lo que es un resultado bastante satisfactorio. En cuanto a los datos obtenidos vemos como se demora unos segundos más cuando se toma la medida en caliente. Cuando se termina el escenario, se borran todos los parámetros de la red y no queda nada cacheado.

En último lugar y el que menos tiempo consume tenemos a Mininet que está por debajo de los 10 segundos, tarda tan poco porque utiliza espacios de red del sistema anfitrión lo que le hace ser muy ligero. No obstante, en este caso no se realiza la activación de rutas de forma dinámica, al carecer Mininet de esta funcionalidad.

En resumen, atendiendo a los resultados obtenidos la herramienta que llevaría ventaja sobre el resto sería Mininet ya que es la que tarda menos tiempo en arrancar. Tarda tan poco por lo que se comentó en su implementación con rutas estáticas, entre otros factores. Debemos tener en cuenta esto para elegirlo como la herramienta de emulación más apropiada o no. No nos podemos olvidar de CORE, Imunes y GNS3 que tienen unas resultados muy buenos en esta prueba, y pueden aportar características interesantes en las siguientes pruebas. Con esto, no se descartan VNX, Netkit, y Marionnet, pero en esta prueba han quedado por detrás de los demás programas.

4.3 Consumo de CPU

Para poder hacernos una idea de cuán eficientes pueden llegar a ser las herramientas en el escenario propuesto, vamos a hacer un análisis del consumo de CPU, durante el despliegue y durante unos segundos después de su estabilización. Con esto trataremos de dirimir cuanto porcentaje del procesador está en uso en cada herramienta de emulación, para así ir acercándonos a nuestro objetivo de poder concretar cuál es la que cumple un mejor compromiso entre prestaciones y consumo moderado de recursos del sistema anfitrión.

Para la obtención de los datos se ha utilizado un programa de monitorización del sistema en Linux llamado nmon, que, a modo de resumen, nos permite visualizar varios parámetros del sistema, desde consumo de CPU a consumo de memoria, hasta datos sobre la red. La elección de este programa surge a raíz de la sencillez en apreciar el consumo de CPU ya que nos ofrece de una manera gráfica los valores en porcentaje de utilización (Figura 4-2).

Para esta prueba se han tomado medidas cada dos segundos y se ha hecho una gráfica en la que se comparan las distintas herramientas de emulación bajo estudio. Como en la prueba anterior, se han realizado las medidas bajo el caso denominado "en frío" y "en caliente".

Cabe destacar que el consumo de CPU que nos ofrece la herramienta nmon puede venir dado por tres factores, por el sistema, por el usuario o por encontrarse la CPU en espera. Las herramientas de emulación bajo estudio son gestionados por la CPU de distintas manera según el método que utilizan para iniciarse. Se muestra una captura de salida de nmon:

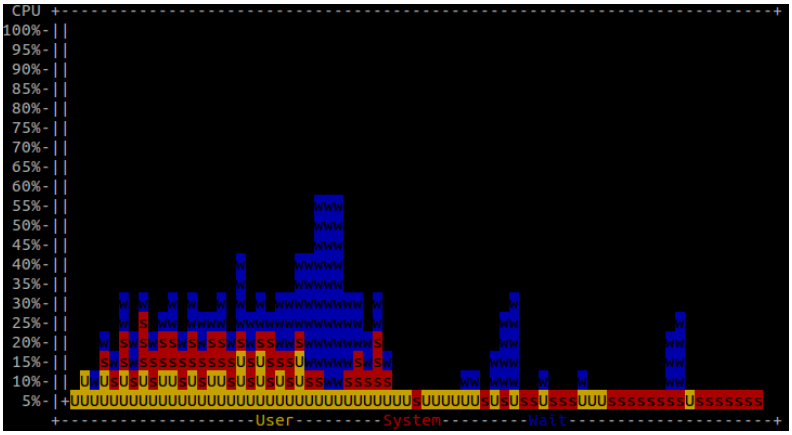

Figura 4-2: Gráfica salida nmon con Marionnet

A continuación se muestran los resultados obtenidos dispuestos en una gráfica para que la comparación sea más sencilla.

4.3.1 Consumo de recursos de CPU (%) "en frio"

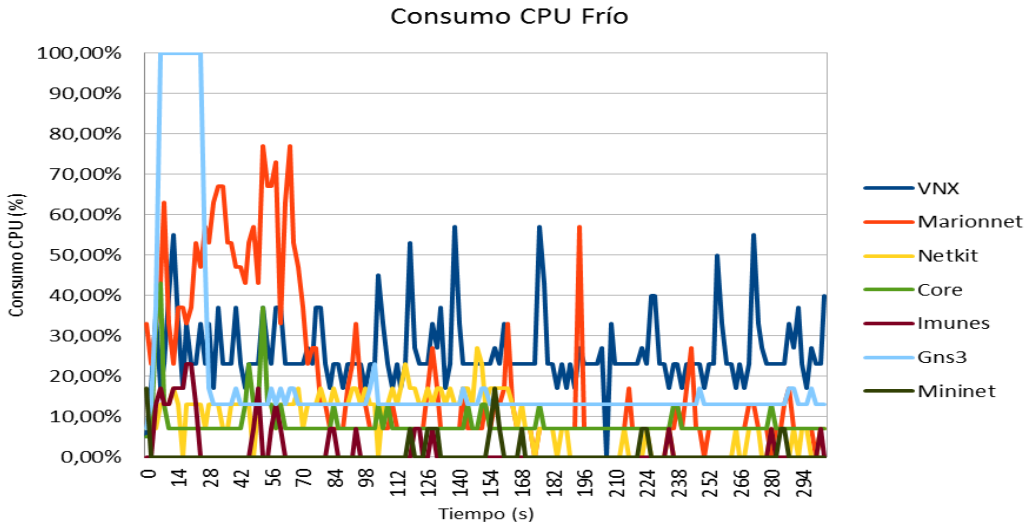

Figura 4-3: Gráfica consumo de CPU en frío

Para comenzar el análisis vamos a centrarnos en las gráficas que presentan un mayor consumo de CPU con respecto a las demás. Estas son la gráficas de GNS3 y Marionnet (azul claro y naranja en la figura 4-3, respectivamente). Se puede apreciar como claramente GNS3 durante el proceso de arranque hasta los 30 segundos aproximadamente consume el 100% de recursos de CPU. Tanto es así que durante el arranque no se puede hacer ninguna tarea en paralelo. Una vez que termina de arrancar todas las imágenes de los *routers*, en un tiempo bastante aceptable, se estabiliza entorno al 15% de consumo de CPU, haciendo algún pequeño pico pero siempre por debajo del 20%. Si tomamos la media total de consumo de CPU en frío nos encontraríamos ante un 19,46%.

Por otro lado, Marionnet es la segunda herramienta que presenta un mayor consumo de CPU durante su proceso de arranque llegando casi al 80%. Una vez que llega a completar el arranque de las máquinas virtuales, salvo algún pico de uso, se estabiliza entorno al 10% de utilización de CPU. Un aspecto curioso de esta herramienta es la manera en la que va iniciando las máquinas virtuales, ya que lo hace de manera sucesiva. Con esto nos referimos a que va arrancando de una en una las máquinas virtuales. Esto se aprecia perfectamente en la salida del comando `nmon`, (figura 4-2), en la cual se ve como la mayoría del tiempo el procesador se encuentra en el estado *wait*. La media de consumo de CPU de esta herramienta es del 19,34%.

En siguiente lugar tendríamos a VNX, que sigue más o menos la misma dinámica durante el arranque, ofreciendo varios picos que coinciden con la inicialización de las máquinas virtuales. El mayor de los picos no sobrepasa en ningún momento el 60% de consumo de CPU lo que en comparación con los otros dos escenarios es una mejoría. El problema de VNX es que el consumo medio es muy alto llegando al 25,88%. VNX consume tanto en media debido a sus virtualizaciones con LXC de una imagen de Linux 14.04. También se probó VNX con KVM, pero se disparaba el consumo del CPU hasta cotas de ser inaceptable ya que el sistema anfitrión no podía arrancar todas las maquinas en todas las simulaciones y el tiempo que requería era muy elevado.
En el margen de menos del 50% de consumo de CPU encontramos a Netkit, CORE, Imunes y Mininet. Vamos a comenzar por CORE. Esta herramienta durante su inicialización alcanza el 42% de utilización de CPU y posteriormente baja notablemente su consumo hasta conseguir estabilizarse entorno entre el 10% y 15%. Puede apreciarse un pico que resalta sobre el resto y se corresponde cuando se arranca el protocolo de *routing* en todos los nodos y se lleva a cabo la comunicación para rellenar las tablas de *routing*. El consumo medio total de esta herramienta es del 8,35%.

Netkit sigue muy de cerca a CORE, siempre estando por debajo del 30% de consumo total de CPU. Fluctúa mucho el consumo de CPU durante el arranque, pero una vez que se estabiliza, baja notablemente variando entre el 0% y el 10%. También durante el proceso de inicialización de las máquinas virtuales se produce el arranque de los protocolos de *routing*, por lo tanto lo hace a la vez, y una vez que están todas las máquinas levantadas se producen los intercambios de paquetes para localizar los nodos adyacentes. Netkit consume como media 8,37% de recursos del CPU.

En último lugar encontramos a las dos herramientas de emulación que utilizan el menor porcentaje de CPU, estas son Imunes y Mininet. Vamos a comenzar analizando Imunes. Durante el proceso de creación del escenario llega alcanzar como máximo menos del 25%, lo que es una cifra bastante buena, no llegando a ser un cuarto del consumo total de CPU. Una vez que finaliza la creación de las máquinas virtuales, pasaríamos a ver otros picos

más pronunciados, sin llegar al 20%, que se corresponden con la generación de las tablas de routing por parte de Quagga. Una vez que ha quedado todo correctamente iniciado, el consumo varía entre el 0% y el 5% (es mínimo). Como media, Imunes presenta el 1,83%.

Por último tenemos Mininet que mirando la figura 4-3 vemos como apenas utiliza recursos de CPU, la cota más alta se encuentra cuando se arranca el escenario y se encuentra entorno al 17% de utilización, después el consumo baja quedando estable en 0% con algun pequeño pico que alcanza el 5%. El escenario creado en Mininet tiene la disparidad con respecto al resto, sobre la implementación de la conectividad entre nodos con rutas estáticas. Esta valoración de no poder establecer rutas dinámicas será valorada negativamente en apartados posteriores. El resultado medio de consumo de CPU es del 0,73%.

4.3.1 Consumo de recursos de CPU (%) "en caliente"

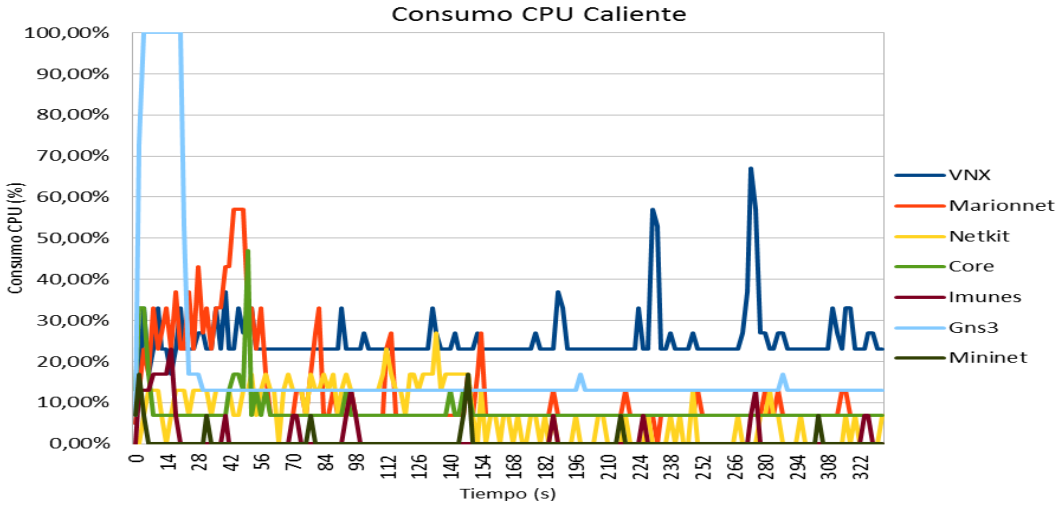

Figura 4-4: Gráfica consumo de CPU en caliente

Observando en la figura 4-4 los datos obtenidos al tomar las medidas una vez que ya habíamos arrancado el escenario previamente, de manera global, los resultados muestran una mayor estabilidad y se ven menos fluctuaciones que en el caso de la primera ejecución. Esto nos puede dar idea de que pueden quedar algunas partes cacheadas y para su arranque es necesaria menos utilización de CPU. Hay que resaltar que para realizar esta gráfica se ha tomado más tiempo que en el caso anterior ya que el programa que más tarda en ejecutarse, aumenta su tiempo en caliente.

Igual que en el caso de las medidas en frío hay una herramienta que destaca por alcanzar el consumo del 100% de CPU pero al cabo de pocos segundos, una vez que consigue crear todas las imágenes de los *routers* se estabiliza en torno al 15% de consumo total de CPU. Esta herramienta es GNS3, que en caliente tiene una media de utilización de CPU del 18,37%.

Marionnet presenta una mejora durante el arranque, y ha pasado a llegar a un máximo por debajo del 60%, cuando en el primer arranque estaba próximo al 80%. Una vez que llega a estabilizarse el escenario ronda el 7% de consumo de CPU, donde también se aprecia una mejoría con respecto a la medida tomada en frío. El consumo medio de CPU de Marionnet se sitúa ahora en el 12,44%. De igual manera tiene la característica que el resto no tiene,

que deja gran porcentaje de CPU en espera (estado *wait*) mientras arrancan las máquinas virtuales, cosa que en el resto es mínimo el tiempo en *wait* y está gestionado totalmente por el sistema (*system*).

En VNX se aprecia una mejora en los primeros segundos del inicio del escenario, ya que el consumo es moderado, entre el 30% y el 40%. Se pueden apreciar dos picos muy pronunciados que son provocados cuando se arrancan los servicios de *routing*, en el caso de frío también se pueden apreciar, pero resaltan más en este caso, ya que el consumo es relativamente constante hasta llegar a ese punto. La utilización media en este caso de VNX es del 25,13%, que es similar al caso de la medida en frío, no se aprecia una gran mejoría. El consumo de CPU está repartido entre el *user* y el *system*, no se pone, en casi ningún momento, en estado *wait*.

Ahora volvemos a encontrar a los mismos programas en el rango que está por debajo del 50% de consumo: son Netkit, CORE, Imunes y Mininet. No hay gran diferencia con respecto a los resultados obtenidos en la primera medida pero sí se puede apreciar algún cambio. Por ejemplo, CORE en los primeros segundos una vez arrancado no presenta un consumo tan elevado como la medida en frío, pero pasado ese tiempo sí que se aprecia una subida del consumo, que en frío también aparece, solo que en esta segunda medida mucho más pronunciado alcanzando el 45% de consumo. CORE presenta un consumo medio en caliente de 8,05% que es algo inferior a la medida tomada en frío, que era del 8,35%.

Netkit sigue una dinámica prácticamente igual que en el caso de la medida en frío, se podría destacar que se aprecia muy bien que cuando se crea cada máquina virtual se activan los servicios que propician la conectividad, ya que cuando están todas las maquinas creadas, el consumo decae situándose entre el 0% y el 10%. El consumo medio de CPU de Netkit es de 7,08%, el cual sí que ha mejorado en un 1% al realizar las pruebas una vez que el escenario ya ha sido previamente probado.

En cuanto a Imunes, durante el arranque se comporta exactamente igual que en la medida realizada inicialmente pero sí que se aprecia que aparecen menos picos secundarios durante la estabilización del escenario. En este caso no se aprecia tan bien el momento en el que se arrancan los servicios que darán conectividad a la red. De igual manera que en el dato adquirido en la medida en frío, el máximo punto de consumo se sitúa en el 25%. El consumo medio de Imunes 1,40%, prácticamente similar al obtenido en la medida anterior.

Por último, e igual que en el caso anterior tenemos a Mininet como la herramienta que menos recursos de CPU consume. Tiene validez todo lo que se comentó en el análisis de Mininet anterior ya que se ha realizado la implementación del escenario con rutas estáticas, en vez de rutas dinámicas como se ha hecho en el resto de escenarios. La media de Mininet sigue siendo la más baja y se sitúa en un 0,71%.

Para concluir este subapartado vamos a resumir las conclusiones obtenidas teniendo en cuenta todas las consideraciones mencionadas. En primer lugar, de manera general, vemos como se ha apreciado una mejoría en las medidas tomadas cuando ya se había iniciado el escenario antes. Esto, nos indica que puede haber procesos que queden cacheados de los cuales pueda servirse la CPU en situaciones futuras para crear el escenario economizando algún recurso ya que ha sido creado con anterioridad.

También se ha destacado que había dos grandes segmentos de programas, aquellos que sobrepasaban el 50% de consumo en algún momento de la ejecución de la topología y otros que se encontraban por debajo. De los del primer segmento veíamos cómo cuando una vez que terminaban de arrancar todas las máquinas virtuales pasaban a tener un consumo moderado, aun así, más elevado que las del segundo segmento.

Por lo cual, las herramientas del segundo segmento son las que obtienen mejores resultados en este apartado, para posteriormente, identificar cual será la que presente mayor compromiso entre prestaciones y rendimiento.

En el anexo C se muestra la figura C-1 que presenta el consumo medio de CPU obtenido en frío y en caliente en las diferentes herramientas.

4.4 Consumo de memoria

Una vez analizado el comportamiento de la CPU durante el despliegue y ejecución del escenario bajo estudio, pasamos al siguiente banco de pruebas realizado, que se corresponde con el consumo de memoria de cada herramienta durante el arranque y unos breves segundos posteriores a la estabilización de la topología simulada. Con este *benchmark* podremos determinar qué herramienta utiliza un mayor porcentaje de memoria durante la ejecución del escenario que estamos analizando (red NSFnet).

Para la realización de esta prueba hemos utilizado el comando `free`, también nativo de Linux, el cual nos reporta la memoria total del sistema, la memoria consumida y la memoria libre entre otros valores que a efectos de esta prueba no vamos a tener en cuenta. Para seguir el mismo criterio que para la prueba anterior se han tomado medidas cada dos segundos, para ello se ha realizado un *script* con el cual sacábamos un informe de la memoria con el comando `free` cada dos segundos y almacenábamos la salida en un fichero de texto para posteriormente analizar y agrupar dichos datos.

Como en casos anteriores se han tomado los datos del comando `free` en las dos situaciones de interés, recién encendido el sistema anfitrión y después de haber iniciado el escenario.

4.4.1 Consumo de memoria “en frío”

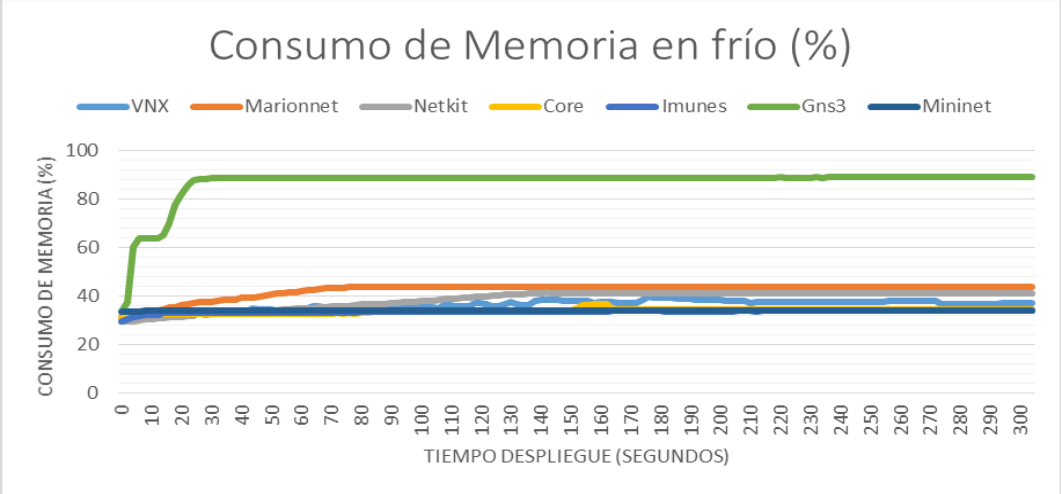

Figura 4-5: Gráfica consumo de memoria en frío

Con una primera vista de la figura 4-5 se puede apreciar rápidamente cuál es la herramienta de emulacion que requiere un mayor porcentaje de memoria. Esta herramienta es GNS3, pero no solo consume mucha memoria, si no que lo hace de manera prolongada permaneciendo inmutable. Se observa cómo, al principio, consume un porcentaje morderado pero cuando estan todas las virtualizaciones activas sobrepasa el 80% de consumo de memoria, lo que es bastante elevado. Realmente GNS3, en este apartado tiene la partida perdida con respecto al resto de herramientas de virtualizacion que se encuentran entorno al mismo porcentaje que va desde el 30% al 45%.

El segundo grupo de herramientas es aquel que engloba todos menos GNS3, presenta un consumo de memoria bastante razonable. Vamos a analizar dentro de este grupo cual puede ser el que nos beneficie más, con un consumo menor.

Marionnet se desmarca dentro del segundo grupo ya que se puede apreciar en la gráfica como es la que presenta un mayor consumo de memoria, que se encuentra un poco por encima del 42% del total de la memoria, que pese a ser el más elevado es totalmente aceptable, por lo tanto el resto solo mejoraran este resultado.

En segundo lugar con un mayor consumo de memoria aparace Netkit con un 42% del total de memoria del sistema anfitrión. Netkit utiliza para virtualizar UML lo que como podemos apreciar, provoca un consumo de memoria algo elevado. Aún así es un consumo bastante razonable. En la gráfica se puede apreciar como comienza con un consumo bajo pero a la hora en que las maquinas virtuales se levantan va creciendo, como cabría esperar, el consumo de memoria.

Llega el turno de VNX, cuyo consumo de memoria esta ya ligeramente por debajo del 40% del total. De igual manera que en los otros escenarios, a medida que se van iniciando las máquinas virtuales, en este caso por medio de LXC, va aumentando el porcentaje de consumo hasta llegar a estabilizarse. Podemos llegar a la conclusion que LXC virtualiza de manera algo más ligera que UML, aunque hay otros factores de por medio.

En ultimo lugar vamos a analizar las tres herramientas de emulación restantes. Dichas herramientas tienen un porcentaje de consumo de memoria muy parejo, tanto es asi que en la grafica presentada se solapan unas con otras. Dichas herramientas son Imunes, CORE y Mininet. Podemos ir dándonos cuenta de cuáles son los programas que aparecen en las mejores posiciones porque reunen unas características comunes en cuanto a rendimiento. Estas tres últimas herramientas se encuentran entre el 30% y el 35% de consumo de memoria del sistema anfitrion.

Como en el resto de escenarios, a medida que se van iniciando las maquinas virtuales del escenario el consumo de memoria va aumentando. Mininet es el que muestra una respuesta más constante desde su inicio hasta su fin. En CORE aparece un pico por encima del resto que no se corresponde con el inicio de los protocolos de red para que haya conectividad entre los nodos ya que dicho servicio se inicia bastante antes. Por lo que puede ser por cualquier proceso interno del sistema anfitrion ajeno a CORE porque enseguida vuelve a estabilizarse.

Imunes por su parte no presenta ninguna irregularidad notable y su respuesta es bastante buena ya que tiene consumo bastante moderado de la memoria.

Para cohesionar un poco las ideas, nos quedamos en este apartado con las tres últimas herramientas analizas, CORE, Imunes y Mininet. No podemos obviar el defecto de Mininet, que tiene las rutas establecias de manera estática y no utiliza ningún demonio para arrancar los servicios de *routing*, lo que podría hacer que aumentara su consumo de memoria un poco.

4.4.2 Consumo de memoria "en caliente"

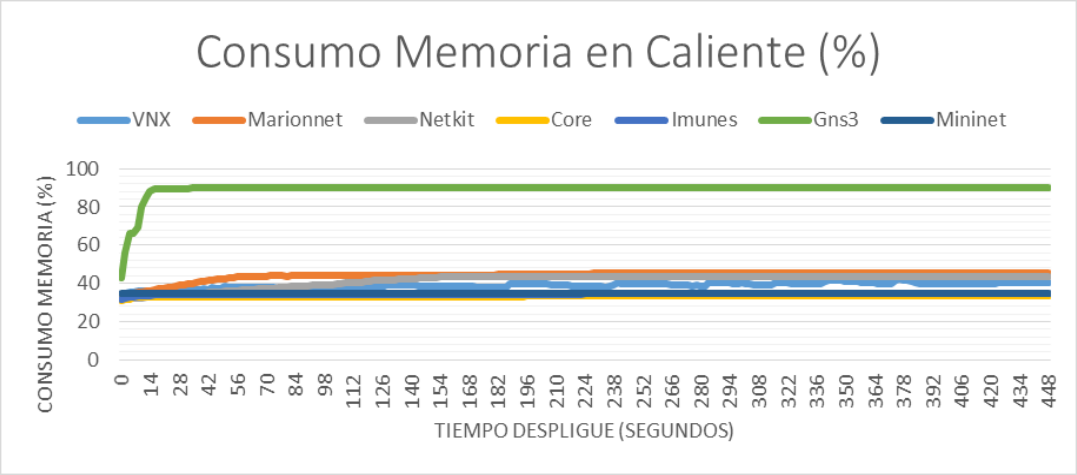

Figura 4-6: Gráfica consumo de memoria en caliente

Antes de comenzar con el análisis de los resultados, es importante darse cuenta que para la obtencion de esta gráfica se ha tomado un intervalo mayor de tiempo, ya que se tuvo que esperar a que el escenario más lento finalizara su ejecución. Como se ha visto en la primera prueba del apartado 4, (4.1 Tiempos de despliegue), había una herramienta que tardaba más en arrancarse en caliente que en frío, de ahí el aumento del número de valores obtenidos.

Para comenzar el análisis con un visión superficial de la figura 4-6, vemos que presenta una dinámica muy similar a la que obtuvimos con las medidas en frío. GNS3 continua presentado un consumo de memoria elevado, rondando el 90% de utilización. También, hay que recalcar el comienzo de la gráfica, que lo hace en torno al 42%, mientras que el valor cuando se iniciaba en frío era del 34%. Esto nos puede dar a entender que a la hora de terminar el escenario queda algún proceso que consume memoria. Por ello, cuando se vuelve a arrancar el escenario, la gráfica parte de un porcentaje de utilización mayor. En esta herramienta no se nota mejoría alguna y sigue siendo muy elevado.

En cuanto a Netkit y Marionnet, son las dos herramientas que siguen presentando un consumo más elevado, siempre por debajo de GNS3, con respecto a las herramientas que presentan un mejor rendimiento en este apartado. Marionnet pasa del 30% de consumo de memoria antes de iniciar el escenario, hasta el 43% una vez que ha arrancado todas las máquinas virtuales y sevicios de comunicación entre nodos de la red. Este resultado permanece igual que en el caso de la prueba tomada en frío. Marionnet parte de un porcentaje de utilización de memoria igual que en el caso de frío, lo que nos da a entender que cuando se finaliza, termina con todos los procesos. Realmente, todos los programas

comienzan desde el mismo punto que en frío por lo que se puede llegar a la misma conclusión, a excepción de GNS3 que ya se ha comentado.

Netkit sí que consume un mayor porcentaje de CPU en caliente, comparándolo con la medida tomada en frío. Hemos pasado de un maximo del 42% ,en frío, a un 43% de utilización en caliente. No es un aumento muy notable, porque efectos de las comparacciones sigue estando por debajo de la mitad de consumo de memoria total.

En siguiente lugar vemos como VNX ve incrementado ligaremante su consumo de memoria, ya que en el caso de la primera medida en frío no sobrepasaba el 40% del total y en este caso sí que lo rebasa. También se aprecia como el consumo sigue una dinámica más constante que en el caso de frío, que se comportaba de manera más aleatoria. Ahora se puede apreciar el proceso de arranque de cada máquina virtual con pequeños picos de consumo. Sigue manteniendo un consumo asumible.

En último lugar del análisis tenemos las tres herramientas que continúan con un consumo bastante moderado de memoria. Son CORE, Imunes y Mininet. Dichas herramientas están alrededor del 35% de porcentaje de uso de memoria. Mininet es el que se posiciona un poco por encima en cuanto a consumo respecto las otras dos herramientas de emulación, ya que esta sobre el 35% de utilización. En el caso de la medida tomada en frío presentaba el mismo comportamiento, y no se aprecia ninguna mejora ni empeoramiento al realizar esta prueba.

Imunes se encuentra situado a medio camino entre el consumo de mininet y el de CORE, su consumo de memoria es de 32% más o menos. Mantiene el consumo moderado que ya presentaba en medidas anteriores.

Para finalizar el análisis, CORE, es el que consume menor porcentaje de memoria en caliente. En las medidas tomadas en frío no se podía apreciar con claridad cual era la herramienta que se desmarcaba, teniendo un consumo menor, pero en este caso sí que se puede determinar que es CORE porque la gráfica obtenida lo muestra perfectamente. En la medida en frío se apreciaba una irregularidad en el comportamiento que no se correspondía con un proceso del programa, se concluyó que era algún tipo de proceso del sistema del anfitrión que requirió, durante un breve instante, un pequeño cosumo de memoria. Y ahora al analizar la gráfica de CORE ratifica lo que pensábamos por que en esta ocasión aparace una respuesta totalmente uniforme.

A modo de pequeño resumen de este subapartado se puede comentar cómo algunas herramientas han visto su comportamiento alterado, como ha sido el caso de GNS3, mientras que otras permanecían con la misma dinámica de las medidas anteriores. Si hubiera que destacar alguna herramienta por encima del resto, esa sería CORE, ya que se aprecia como claramente su consumo de memoria es el más bajo.

En el anexo C se muestra la figura C-2 que presenta el consumo medio de memoria obtenido en frío y en caliente en las diferentes herramientas.

4.5 Ancho de banda

La última prueba que se va a realizar sobre las herramientas de emulación seleccionadas es aquella que implica el ancho de banda que podemos alcanzar en ellas. El ancho de banda se define en este caso como la cantidad de datos que se transmiten a través de una

conexión de red durante un período de red establecido. Esta medida es importante para ver cómo actuarían las herramientas y la facilidad que nos ofrecerían para hacer medidas que podrían realizarse en una implementación real. En este caso como toda la red simulada está en un único sistema anfitrión, estas medidas dependerán del bus de memoria del mismo, ya que las conexiones entre las máquinas virtuales son todas lógicas y no físicas.

Hay que resaltar que algunas herramientas permiten limitar el ancho de banda de los enlaces que se han creado entre los nodos, pero, para esta prueba, no se ha tocado ese parámetro y se ha supuesto que todas tienen el máximo posible sin ningún tipo de limitación. Esto se tendrá en consideración en el próximo apartado.

Para la realización de esta prueba nos hemos apoyado en un comando básico para el estudio de ancho de banda en sistemas Linux. Este comando es `iperf`, el cual nos permite medir el ancho de banda entre dos *hosts*. Se caracteriza por ser una herramienta cliente-servidor. El procedimiento para medir el ancho de banda es sencillo. En primer lugar se identificaría el camino entre el cual se quiere determinar el ancho de banda, en nuestro caso son los siguientes caminos:

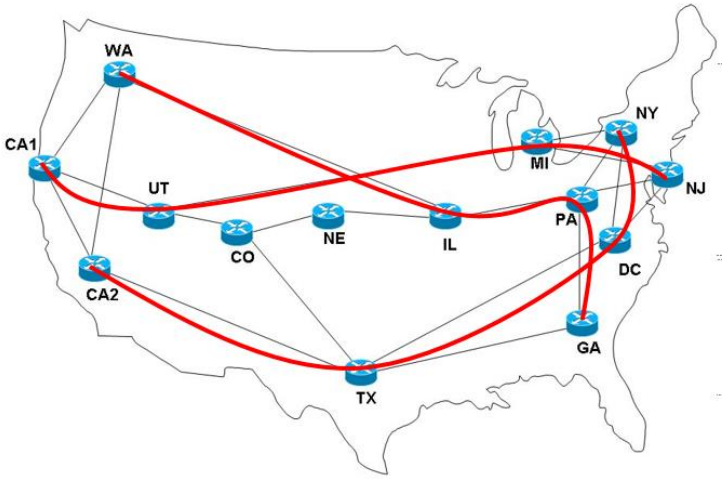

Figura 4-7: Rutas para medir ancho de banda

Como se ve en la figura 4-7 todos los caminos tienen el mismo número de saltos, y no se solapa ninguno, para que las medidas sean lo más equitativas posibles.

Una vez que tenemos identificadas las rutas a seguir, nos aseguramos con un `traceroute` que realmente es ese el camino que siguen. Una vez hechas estas consideraciones iniciales podremos comenzar a analizar el ancho de banda de la red. Para ello, el siguiente paso sería arrancar el servidor que estará pendiente hasta que establezca la conexión el cliente.

El comando utilizado para realizar las medidas es el mostrado a continuación, en el que se detalla por qué se han determinado esos parámetros.

```
root@GA:/# iperf -s -P 0 -i 10 -w 4128.0B -M 536.0B -l 8192.0B -f MB
```

Figura 4-8: Comando *iperf* servidor

Los parámetros que aparecen se han escogido de la siguiente manera:

El `-s` indica que no encontramos en la parte del servidor, `-P` indica streams en paralelo, `-i` el intervalo de reporte, `-w` el tamaño de la ventana en bytes, `-M` el MSS (*maximun segment*

size), `-l` tamaño del buffer y por último el `-f` indica el formato de la salida (MB=Mega bytes)

```
root@WA:/# iperf -c 10.0.12.1 -P 1 -i 10 -w 4128.0B -M 536.0B -l 8192.0B -f MB -t 10
```

Figura 4-9: Comando *iperf* cliente

Donde los parámetros son similares a los del servidor a excepción que se pone el `-c` para indicar que es el cliente, la dirección IP de destino para que se establezca la conexión.

Esto nos daría una salida del siguiente estilo:

```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 0.01 MByte (WARNING: requested 0.00 MByte)
------------------------------------------------------------
[  4] local 10.0.12.1 port 5001 connected with 10.0.18.1 port 46575
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  1062 MBytes   106 MBytes/sec
[  4]  0.0-10.0 sec  1062 MBytes   106 MBytes/sec
```

Figura 4-10: Salida comando *iperf*

La elección de esos parámetros viene a raíz de que en el programa GNS3 no se puede utilizar el comando `iperf`, ya que utiliza unas imágenes de unos *routers* de un fabricante en concreto y tiene sus propios comandos. En GNS3 se ha utilizado el comando `ttcp` de la manera que sea lo más similar a `iperf`, de ahí la elección de esos parámetros.

Se han recopilado todos los datos obtenidos en la figura 4-11 que se presenta a continuación, donde se ha situado para cada camino el ancho de banda de todas las herramientas.

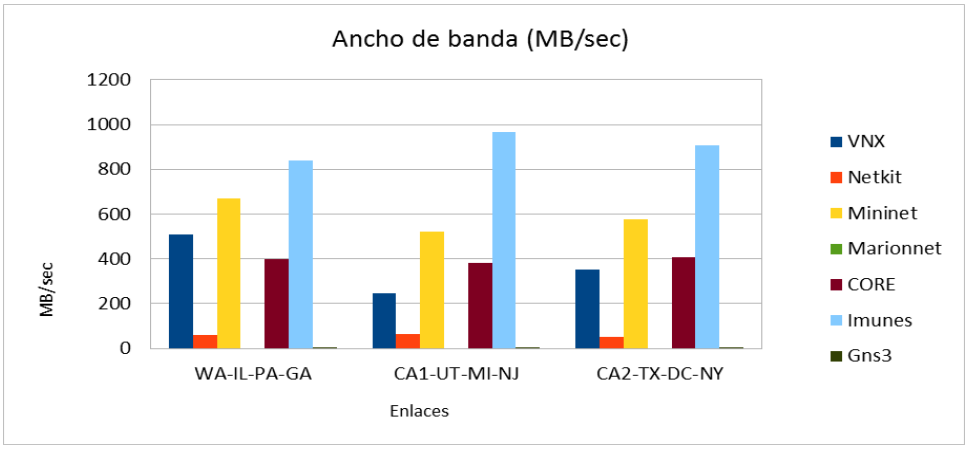

Figura 4-11: Gráfica consumo de ancho de banda Mb/s

Observando la figura 4-7 vemos como hay una herramienta que lleva ventaja sobre el resto, esta es Imunes, la cual, en las diferentes rutas presenta un ancho de banda superior a las demás herramientas. El máximo ancho de banda que alcanza es de unos 1000 Mb/s, hay que recordar que este valor puede ser mucho mayor ya que, para esta prueba, se han establecido unas limitaciones para que el estudio con `ttcp` sea equitativo. El siguiente que tiene un ancho de banda mayor es Mininet, el cual tiene las rutas establecidas de manera estática, solo como recordatorio, aunque a efectos de esta prueba es indiferente si la conectividad se establece mediante rutas estáticas o dinámicas. El máximo de Mininet se encuentra ligeramente por encima de 600 Mb/s .

En siguiente lugar la herramienta que presenta un ancho de banda mayor es VNX, aunque no lo hace de manera constante. Tiene el tercer mayor ancho de banda pero no lo mantiene en los tres recorridos establecidos si no que, en el primer recorrido presenta un ancho de banda 500 Mb/s pero en los otros dos queda por debajo de CORE, que muestra un ancho de banda más estable, entorno a los 400 Mb/s.

En última posición en esta prueba estarían Netkit y GNS3 que como se ve en la figura 4-11 muestran los anchos de banda más bajos. Netkit con un ancho de banda máximo de 64 Mb/s y GNS3 con un ancho de banda máximo de 4.92 Mb/s.

Como se habrá observado en esta prueba no se ha incluido a Marionnet, ya que no ofrecía la posibilidad de utilizar ningún método para medir el ancho de banda. Se ha tratado de instalar `iperf`, pero no lo admitía. Además, se investigó y no había ninguna solución alternativa. Por tanto no obtiene ningún punto a favor en este apartado.

También se han tomado las medidas de ancho de banda sin la limitación que imponía `ttcp`, para así ver cuánto es el máximo que podemos llegar a obtener, como curiosidad. Se ha excluido de esta medida a GNS3 y a Marionnet por no soportar este comando. A continuación se muestra la figura 4-12:

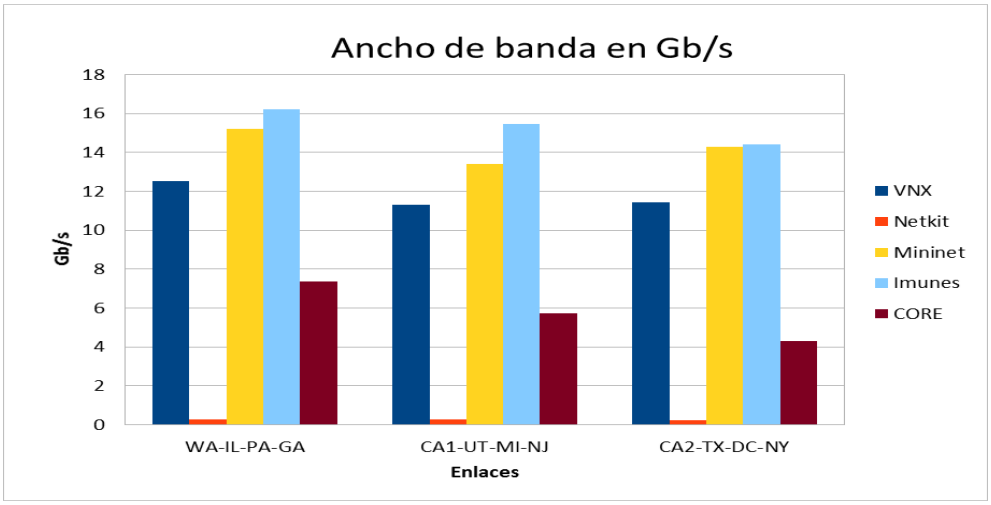

Figura 4-12: Gráfica consumo de ancho de banda en Gb/s

En la figura 4-12 se puede apreciar como los resultados obtenidos son más estables. Comparándolo con la figura 4-11, VNX sigue por detrás de Imunes y Mininet, pero ahora sí que supera en todas las rutas a CORE.

4.6 Otras características

En este último apartado, previo a las conclusiones finales, vamos a analizar las prestaciones en el diseño que nos puede ofrecer cada herramienta de emulación analizada anteriormente. Por ejemplo, en este apartado entrarán en juego la facilidad de creación de los escenarios, los diferentes tipos de virtualización que permite la herramienta, la sencillez que presentan a la hora de añadir servicios a la red y meter en ella retardos, fallos de *routers*, fallos en las líneas de transmisión de datos etc. También sería importante la semejanza con escenarios que pudieran darse en la vida real. Para ello se ira analizando

una a una todas las herramientas destacando o indicando las ventajas e inconvenientes que está presente, fijándonos básicamente en los parámetros comentados anteriormente.

Vamos a comenzar por VNX. Un aspecto muy bueno que tiene a su favor, es que permite numerosos métodos de virtualización. Entre ellos se encuentran KVM, UML, LXC, Olive y dynamips, cado uno tiene sus ventajas e inconvenientes. En cualquier caso, tener la disponibilidad de diseñar un escenario que englobe distintas virtualizaciones es un punto a su favor muy importante, puesto que en una red en la vida real pueden encontrarse sistemas muy dispares, entre hosts, servidores etc. Además, tiene un repositorio desde el cual se pueden descargar distintos sistemas de archivos para poder emularlos: podemos encontrar distintas versiones de Ubuntu, Fedora, Debian, Android, freebsd, kali... Otro aspecto que tiene a su favor es que la implementación del escenario es muy intuitiva. Para la descripción del escenario hay que crear un script en XML en el cual se detallan los enlaces físicos que habrá que generar entre los nodos, así como los nodos que habrá en la red y el tipo de virtualización que se utilizara. Por experiencia propia, sin tener un conocimiento muy avanzado sobre XML y únicamente guiándonos con los tutoriales que presentaba la herramienta hemos podido implementar la topología sin mucho problema, lo que también es un punto a importante a considerar. A la hora de la conectividad entre los nodos, los desarrolladores de VNX tienen un *script* de ejemplo de configuración que permite la utilización de OSPF. Con unas ligeras modificaciones se puede conseguir que se despliegue en la red. No habría problemas para desplegar otro protocolo de *routing* ya que el *script* realmente de lo que se encarga es de arrancar Quagga, el cual soporta una amplia variedad de protocolos de *routing*. Permite apagar máquinas virtuales del escenario sin detenerlo completamente, lo que equivaldría a que se cayera un nodo de la red. Entonces, se podría medir cuán problemático sería, ya que gracias al protocolo de *routing* se recalcularían las rutas. Entre los aspectos negativos encontraríamos que no se puede establecer ningún tipo de parámetro en los enlaces entre *routers*, ya que la herramienta no nos da esa opción. Pero sí se puede, si se hace aparte, en el anfitrión, usando `tc` y `netem`.

En segundo lugar Netkit, no presenta tantos métodos de emulación como la herramienta analizada en el párrafo anterior. De hecho utiliza solo UML, por lo que únicamente soporta máquinas virtuales que corran en Linux, lo que ya limita un poco a Netkit si lo comparamos con una red real en la que aparecen distintos tipos de sistemas operativos en las máquinas. Para la implementación de este escenario no se necesita conocimiento de programación de ningún tipo. Esto se debe a que está basado en crear una serie de archivos de configuración, los cuales, el intérprete de Netkit se encarga de traducir y generar la topología. Estos archivos de configuración son sencillos, uno de ellos es aquel que establece cómo serán las conexiones entre los nodos y qué interfaz de cada *router* se corresponde con cual. Después habría que hacer un archivo de configuración por cada nodo de la red en el que se define los parámetros de red (direcciones IP) y que se arranque el protocolo de *routing* deseado. Es un proceso muy sencillo, más incluso que en el del apartado anterior, lo que le da puntos a favor. Para establecer los protocolos de *routing* que crearan las rutas para la conectividad entre los nodos de la red emulado, únicamente hay que copiar los archivos de configuración de OSPF, en nuestro caso, pero soportaría cualquier otra ya que se basa también en Quagga. Es muy sencillo arrancarlo ya que en el archivo de configuración de cada máquina se puso explícitamente que arrancara el servicio. En esta herramienta se pueden simular *hosts*, *routers* y *switches*, principales elementos de una red de comunicación. De igual manera se puede comprobar el efecto que tendría que un nodo o varios nodos de la red estuvieran caídos, ya que el programa nos permite finalizar independientemente máquinas virtuales sin afectar al funcionamiento de

la ejecución del escenario. Esta herramienta de emulación también carece de libertad a la hora de modificar el tipo de enlaces. No permite limitar el ancho de banda, ni introducir pérdidas y ni mucho menos establecer retardos en las líneas de transmisión.

Imunes se basa para la emulación en Docker de los elementos de capa 3 como son los *routers*, y para los elementos de capa 2 (*switches*) lo hace mediante OpenVSwitch. Permite al usuario utilizar cada nodo de una manera más realista, sin tener que estar pendiente de la creación ni la sobreescritura de sistemas de archivos del sistema anfitrión, gracias a Docker. El punto que destacaría frente a los que hemos visto hasta ahora es la sencillez para implementar cualquier tipo de escenario. Es una herramienta basada en un entorno gráfico, por tanto para comenzar el escenario únicamente tendríamos que arrastrar los elementos de la red que queramos incorporar. En nuestro caso han sido únicamente *routers* y enlaces, pero también ofrece la posibilidad de añadir *hosts, switches*. En Imunes para arrancar el protocolo de encaminamieto únicamente basta con hacer clic derecho sobre los nodos de la red y en la parte de *configuration* podemos elegir que la creación de rutas sea dinámica, con Quagga, o estática. Una vez que escogemos que se haga de manera dinámica nos da la opción de elegir entre varios protocolos (Los mismo que en los anteriores, RIP, RPIng, OSPF...). Simplemente marcaríamos el que deseamos. Como se puede ver es una implementación bastante rápida y sencilla. Una característica que introduce esta herramienta a diferencia de las analizadas previamente es que sí permite establecer de forma nativa parámetros en los enlaces que unen los *routers*. Además, es muy sencillo hacerlo, hay que hacer clic sobre el enlace en el que queramos configurar alguna limitación y añadirla. Nos permite limitar el ancho de banda, establecer un retardo en la línea, establecer un BER (*Bit Error Ratio,* Tasa de Error Binario) que hace referencia a la cantidad de bits recibidos de forma errónea respecto al total de bits transmitidos durante un intervalo de tiempo. Estos aspectos se valoraran muy positivamente en este apartado ya que las redes reales tienen imperfecciones y poder modificarlo en una simulación puede hacer que nos acerquemos un poco más a esa realidad que buscamos. Un defecto encontrado en Imunes es que una vez que se arranca el escenario no se puede detener un máquina virtual independiente, solo se puede detener todo el escenario. No se podría ver en esta herramienta el efecto que tendría que fallara un nodo de la red, aunque una solución sería eliminarlo de la topología y simularlo. Como los tiempos de ejecución son bastante bajos es totalmente aceptable esta solución.

Una herramienta muy similar a Imunes es CORE, el cual se caracteriza por utilizar Linux Containers (LXC) como método de virtualización, lo que le limita únicamente al uso de un único tipo de virtualización para la creación de las distintas máquinas virtuales del escenario simulado. CORE se caracteriza por tener una interfaz gráfica bastante intuitiva y limpia. La creación del escenario es muy sencilla y no se necesita ningún tipo de conocimiento exhaustivo previo de la herramienta. Esta herramienta presenta la posibilidad de añadir a las topologías *routers*, *switches*, *hosts*, y también redes inalámbricas (que en herramientas anteriores no aparecía). En esta herramienta es muy sencillo activar el protocolo de *routing*, ya que tiene una pestaña llamada *widgets* en la que nos permite elegir el tipo de adyacencia que queramos para los nodos. Por defecto utiliza Quagga como plataforma de *routing*, que tiene los mismos protocolos que las herramientas anteriores, pero CORE también permite la opción de utilizar otras plataformas de *routing* como es XORP, BIRD entre otras herramientas muy útiles, como son SSH, FTP, HTTP, DHCP... Un aspecto a recalcar sobre esta herramienta es que provee de servicios de seguridad: tiene la opción de crear un VPNClient y consecuentemente un VPNServer, también permite IPsec y por último y más importante la opción de que un *router* actúe como *firewall* de

forma nativa. Este último detalle nos permitiría crear una red con todo tipo de elementos, como los que compondrían una red en la realidad, lo que en consecuencia se obtendría una aproximación muy fiel. Este aspecto aporta un valor especial a esta herramienta que las otras no presentaban. Otra característica muy útil de esta herramienta es que permite crear redes inalámbricas y móviles, y se puede definir un rango de alcance y hacer un *script* en el que se defina el movimiento de los nodos haciendo que entren y salgan de la red y ver su conectividad. Para conseguir conectividad en estos nodos hay que instalar Quagga MDR. MDR significa *MANET Designated Router*, donde MANET es *Mobile Ad-Hoc Network*. Para terminar el análisis de CORE, este también nos da la opción de modificar varios parámetros de los enlaces. Estos parámetros son, limitar el ancho de banda, establecer un retardo en la línea (en el orden de microsegundos), establecer jitter, introducir un porcentaje de pérdidas y porcentaje de duplicados. La única pega que se le puede poner a este programa es que no permite, una vez que se ha arrancado la simulación, detener un nodo. La solución a esto sería la misma que se ha planteado para Imunes. El tiempo de ejecución de CORE es muy bajo por lo que esta solución es asumible.

En cuanto a GNS3, como se ha comentado en casos anteriores tiene la característica que emula imágenes de *routers* de unos fabricantes en concreto. Lo hace por medio de dynamips (para Cisco) y puede soportar también olive (para Juniper). Si en la implementación real se trabaja con fabricantes muy concretos, GNS3 puede ser una buena opción. Otro factor importante es que es muy sencillo crear cualquier tipo de topología ya que dispone para su implementación de una interfaz gráfica muy sencilla, en la cual únicamente hay que escoger el elemento de la red que queramos utilizar en el experimento y añadirlo. Por todo lo anterior, la implementación de escenarios es un factor favorable de esta herramienta. La conectividad de la red se crea de manera disitnta que en los escenarios anteriores, pero dispone de los mismos protocolos de encaminamiento y alguno más privativo de fabricantes concretos. Esta herramienta permite añadir *routers*, *switches*, *hosts*, y elementos de seguridad en la red, lo que le hace ser una herramienta muy completa en este ámbito. No permite la opción de modificar parámetros en las líneas de transmisión de datos, lo que le hace perder puntos. Por último, una vez arrancado el escenario, sí permite detener nodos independientes para comprobar cómo afectaría una pérdida de servicio en cualquiera de ellos.

Mininet, por sus características de desarrollo del escenario puede ser el más complicado ya que hay que tener alguna noción básica de Python y hay que estudiar la API de Python que ofrecen los desarrolladores para poder lograr crear la topología deseada. Mininet pierde puntos en cuanto a la sencillez del desarrollo ya que es el que más tiempo ha consumido y no se ha podido conseguir establecer un protocolo de *routing*. No se ha podido establecer ya que se ha intentado hacerlo con Quagga, que teóricamente es totalmente factible. Se ha probado con varios métodos de conseguirlo y no se ha logrado por lo que al final se decidió establecer las rutas de manera estática. Esto hace que la comparación no sea totalmente justa pero para hacernos una idea puede ser útil. Mininet pierde muchos puntos en ese ámbito. Mininet permite añadir *switches* y *hosts* en el inicio, pero cabe la posibilidad de crear unas clases en Python que actúen como *routers* ya que se le activa el *IPFORWARD*. Aquí también Mininet va un paso por detrás que el resto. En cuanto a los enlaces, se definen en el script de Python y no permiten la modificación de ningún parámetro. Mininet no permite detener una máquina virtual de manera individual, habría que detener el escenario y modificar el *script* de Python para simular que un nodo se ha caído. El tiempo de simulación de Mininet es muy bajo, por lo que podría ser una solución.

Por ultimo vamos a terminar indicando las características principales de Marionnet, el cual presenta tres tipos de imágenes para virtualizar, todas ellas basadas en el kernel de Linux. Tiene una imagen específica para los *routers* y otras para los *hosts*. Esta herramienta aporta una manera sencilla de implementar el escenario ya que al igual que algunos programas ya analizados previamente presenta una interfaz gráfica. Esta es algo menos sencilla que las anteriores pero se entiende con facilidad como funciona, por lo que no es un problema. Esta herramienta permite la utilización de Quagga como plataforma de *routing*, por lo que admite varios tipos de protocolos. Permite arrancar y detener nodos independientemente de que todo el escenario este arrancado, lo cual es un aspecto favorable para hacer pruebas de perdida de conexión con algún nodo. También otra característica a favor es que permite modificar de forma nativa parámetros de los enlaces como son pérdidas, retardos etc.

A continuación, en la figura 4-13 se muestra un resumen de todos los aspectos que se han comparado en cada una de las herramientas. Aparece un tic verde cuando presentan dicha característica o una cruz roja cuando carece de ella:

	Programas						
Características	VNX	Netkit	GNS3	Imunes	CORE	Mininet	Marionnet
Varios métodos de virtualización	✓	✗	✓	✗	✗	✗	✗
Modificación enlaces	✗	✗	✗	✓	✓	✗	✓
Sencillez de la implementación	✓	✓	✓	✓	✓	✗	✓
Protocolos de routing	✓	✓	✓	✓	✓	✗	✓
Añadir elementos de seguridad	✗	✗	✓	✗	✓	✗	✗
Soporte wireless	✗	✗	✗	✗	✓	✗	✗
Detener nodos independientes	✓	✓	✓	✗	✗	✗	✓

Figura 4-13: Comparación características

4.7 Conclusiones

Una vez finalizado el capítulo 4 ya tenemos perspectiva de cómo se comporta cada herramienta en cuanto a rendimiento en el ordenador anfitrión se refiere, puesto que se ha analizado el consumo de CPU y de memoria de cada herramienta. También se han visto las características principales que tienen cada herramienta y lo que las diferencia del resto. Se han resaltado las características positivas y las negativas. Estas características, a modo de resumen han sido: el tiempo que tarda la herramienta de emulación en desplegar el escenario creado, el ancho de banda que permite cada herramienta, los métodos de virtualización que permite la herramienta, si permite la manipulación de las características de los enlaces entre nodos, sencillez en la implementación de escenarios, si ofrece la posibilidad de utilizar distintos protocolos de *routing*, si permite añadir elementos adicionales a la red (como *firewalls*), si soportan simulaciones que involucren redes inalámbricas y si se pueden detener nodos independientes de la red mientras la ejecución del escenario sigue activa.

Todo ello para poder llegar a la conclusión de cuál es la herramienta que presenta un consumo moderado de recursos y que pese a ello nos ofrezca gran variedad de prestaciones, y por consecuencia es la más apropiada para la simulación de una red real con gran fiabilidad. En el siguiente capítulo se determinara cual es la herramienta que resulta más favorable en vista a los resultados de las pruebas y las prestaciones que nos ofrece.

5 Conclusiones y trabajo futuro

5.1 Conclusiones

Para finalizar, una vez hecho todo el recorrido de conocer los principales métodos de virtualización, haber estudiado las principales herramientas de emulación *open-source* en Linux, haber implementado en todas ellas el escenario bajo estudio, haber desarrollado una serie de bancos de pruebas y haber obtenido conclusiones, podemos afirmar que hemos llegado a la determinación de cuál de las herramientas bajo estudio es la que se sitúa en una mejor posición con respecto al resto. Para la decisión de esta herramienta han entrado en juego varios factores, los principales son los que se muestran en esta memoria. Para resumir, todos los factores que han llevado a la decisión final han sido, comenzando por el principio, la sencillez en instalar la herramienta, la facilidad para la implementación de los escenarios, la viabilidad de utilizar distintos protocolos de *routing*, el rendimiento asumiendo un consumo moderado de recursos, la libertad en el diseño y las prestaciones de cara al ámbito de la educación o empresarial.

A lo largo de las pruebas se ha ido acotando qué programas iban presentando mejores resultados. Fijándonos en tiempos de despliegue descartamos VNX, Netkit y Marionnet ya que había otras herramientas que estaban por debajo de los sesenta segundos.

En lo que consumo de CPU respecta nos quedamos con Imunes, CORE y Mininet y por consumo de memoria los que mejor rendimiento presentaban eran otra vez, CORE, Imunes y Mininet.

Los que, de media, presentaban más ancho de banda son VNX, Mininet, Imunes y CORE. Como vamos viendo hay programas que se repiten como los que han obtenido mejores resultados en las respectivas pruebas, así que entre ellos está la herramienta que presenta mejores prestaciones con un rendimiento moderado y en definitiva la elegida.

En cuanto al último apartado los que más prestaciones nos ofrecen son CORE, Marionnet, VNX, GNS3 e Imunes. Fijándonos en la Figura 4-13 se puede apreciar claramente como son los que muestran mayor cantidad de aspectos positivos.

Tras valorar todos estos aspectos y también la experiencia personal al trabajar con cada herramienta, llegamos a la conclusión de que las dos herramientas que ofrecen mejores prestaciones manteniendo un uso razonable de recursos del ordenador anfitrión son CORE e Imunes. Esto se debe a que ofrecen buen rendimiento en cuanto a consumo de memoria y CPU, presentan un tiempo de despliegue bastante bajo, tienen una interfaz bastante intuitiva y sencilla de utilizar con gran cantidad de parámetros para modificar y lograr emular una red lo más parecida a una red que encontraríamos en la vida real.

En conclusión la elección está entre CORE e Imunes, la dos están a la par. La elección final vendría por la aplicación final que tendría. Por ejemplo, si queremos emular redes WiFi, nos decantaríamos por CORE, pero si queremos un implementación como la realizada en esta memoria de TFG, Imunes no serviría perfectamente.

5.2 Trabajo futuro

Una vez finalizado el estudio de las herramientas quedarían pendientes otros aspectos de interés que se pueden enfocar para la realización de trabajos futuros. Durante la realización se han observado las carencias de algunos programas, por lo que podría ser una vía para proseguir con un trabajo futuro. Por ejemplo, en Mininet sería interesante implementar los distintos protocolos de routing. Para ello habría que tener un conocimiento elevado de Python y previo estudio de la API de Mininet.

Otro estudio sería interesante realizarlo con redes WiFi, tomando los programas que nos den esa posibilidad (CORE y un programa alternativo de Mininet llamado Mininet-wifi) y hacer un análisis similar al mostrado en esta memoria.

También se podría tomar topologías y analizar la robustez que tienen frente a cambios o errores y detectar como se podrían mejorar para evitar problemas futuros.

6 Referencias

[1] Emmanuel Lochin, Tanguy P´erennou, Laurent Dairaine, ”When Should I Use Network Emulation?”, Cornell University Library , 28 Jun 2011

[2] Walter M. Fuertes and Jorge E. López de Vergara, “A quantitative comparison of virtual network environments based on performance measurements”, Proceedings of the 14th HP Software University Association Workshop, Garching, Munich, Germany, 8-11 July 2007.

[3] Bob Lantz, Brandon Heller, Nick McKeown, “A Network in a Laptop: Rapid Prototyping for Software-Defined Networks”, Monterey, CA, USA. October 20–21, 2010.

[4] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, Nick McKeown, “Reproducible Network Experiments Using Container-Based Emulation”, Stanford University, Palo Alto, CA USA, December 10–13, 2012

[5] Walter Marcelo Fuertes Díaz, “Evaluación de Plataformas Virtuales para Realizar Experimentos de Medidas en Redes IP”, Trabajo de Estudios Avanzados, Escuela Politécnica Superior, Universidad Autónoma de Madrid, 2007

[6] Harinivesh Donepudi, Bindu Bhavineni, Michael Galloway, “Information Technology: New Generations”, Volume 448 of the series Advances in Intelligent Systems and Computing pp 401-411. 29 March 2016

[7] Anuzelli, G., Files, N., Emulation, P. I. X., Optimizations, M. U., & Emulated, H. C. (2011). Dynamips/dynagen tutorial. Online:" http://dynagen. org/tutorial. htm."

[8] “QEMU, open-source processor emulator”, May 2016. http://wiki.qemu.org/Main_Page

[9] “The User-mode Linux Kernel Home Page”, http://user-mode-linux.sourceforge.net/

[10] “Docker”, https://www.docker.com/

[11] “What’s LXC?”, https://linuxcontainers.org/lxc/introduction/#whats-lxc

[12] Pablo Gil, Gabriel J. Garcia, Angel Delgado, Rosa M. Medina, Antonio Calderon, Patricia Marti, “Computer Networks Virtualization with GNS3”, University of Alicante, April 2014

[13] “Common Open Research Emulator” http://www.nrl.navy.mil/itd/ncs/products/core

[14] Zec, M., & Mikuc, M.” Operating system support for integrated network emulation in imunes. In Workshop on Operating System and Architectural Support for the on demand IT Infrastructure”. January 2004.

[15] Jean-Vincent Loddo, Luca Saiu, “Marionnet: a virtual network laboratory and simulation tool”, Marseille France, March 2008

[16] Maurizio Pizzonia, Massimo Rimondini, “Netkit: Easy Emulation of Complex Networks on Inexpensive Hardware”, Innsbruck Austria, March 2008,

[17] D. Fernández, Virtual Networks over linuX, Center for Open Middleware, feb 2014.

[18] Rogério Leão Santos de Oliveira, Ailton Akira Shinoda, Christiane Marie Schweitzer, Ligia Rodrigues Prete, “Using Mininet for Emulation and Prototyping Software-Defined Networks”, Brazil, June 2014

[19] Fermín Galán “Virtual Network User Mode Linux”, https://www.dit.upm.es/vnumlwiki/index.php/Example-NSF-14, July 1st, 2008

[20] Jakma, P., & Lamparter, D. (2014). Introduction to the quagga routing suite. IEEE Network, 28(2), 42-48.

Anexos

A Plataforma de routing (Quagga)

Quagga es una plataforma de routing que permite la implementación de diferentes protocolos de routing para entornos de Unix. Entre ellos encontramos OSPFv2, OSPFv3, RIPv1 y v2, RIPng y BGP.

La arquitectura de Quagga se basa en zebra que actua como una abstracción del nucleo de Unix subyacente. El esquema sobre el que se fundamente es el siguiente:

Figura A-1: Arquitectura de Quagga.

Quagga se ha utilizado en todas las herramientas a excepción de GNS3 y Mininet por tener sus propios métodos de conectividad.

Archivo de configuración de Quagga:

(Daemons)

```
# This file tells the zebra package
# which daemons to start.
# Entries are in the format: <daemon>=(yes|no|priority)
# where 'yes' is equivalent to infinitely low priority, and
# lower numbers mean higher priority. Read
# /usr/doc/zebra/README.Debian for details.
# Daemons are: bgpd zebra ospfd ospf6d ripd ripngd
zebra=yes
bgpd=no
ospfd=yes
ospf6d=no
ripd=no
ripngd=no
```

(OSPF)

```
!
hostname ospfd
password zebra
enable password zebra
!
! Default cost for exiting an interface is 10
!interface eth0
!ospf cost 21
!interface eth1
!ospf cost 36
!
router ospf
! Speak OSPF on all interfaces falling in 10.0.0.0/16
network 10.0.0.0/16 area 0.0.0.0
redistribute connected
!
!log file /var/log/quagga/ospfd.log
!
```

(Zebra)

```
! -*- zebra -*-
!
! zebra configuration file
!
hostname zebra
password zebra
enable password zebra
!
! Static default route sample.
!
!ip route 0.0.0.0/0 203.181.89.241
!
!log file /var/log/zebra/zebra.log
```

Para utilizar otro protocolo de *routing* únicamente habría que activarlo en la archivo *daemon* y posteriormente crear su archivo de configuración.

B Implementación escenario en VNX y Mininet

A continuación se muestra el código generado para los escenarios que requerían de implementación por scripts:

VNX:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
~~~~~~~~~~~~~~~~~~~~
VNX NSFNET
~~~~~~~~~~~~~~~~~~~~
-->

<vnx xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="/usr/share/xml/vnx/vnx-2.00.xsd">
<global>
    <version>2.0</version>
    <scenario_name>VNX_NSF</scenario_name>
    <automac/>
    <vm_mgmt type="private" network="10.0.0.0" mask="24"
offset="16">
    <host_mapping />
    </vm_mgmt>
     <vm_defaults>
          <console id="0" display="no"/>
          <console id="1" display="yes"/>
    </vm_defaults>
  <extension plugin="ospf" conf="ospfd_conf.xml"/>
  </global>

   <net name="ca1-wa" mode="virtual_bridge" />
   <net name="ca2-wa" mode="virtual_bridge" />
   <net name="il-wa" mode="virtual_bridge" />
   <net name="ca1-ca2" mode="virtual_bridge" />
   <net name="ca1-ut" mode="virtual_bridge" />
   <net name="ca2-tx" mode="virtual_bridge" />
   <net name="mi-ut" mode="virtual_bridge" />
   <net name="co-ut" mode="virtual_bridge" />
   <net name="co-ne" mode="virtual_bridge" />
   <net name="co-tx" mode="virtual_bridge" />
   <net name="il-ne" mode="virtual_bridge" />
   <net name="il-pa" mode="virtual_bridge" />
   <net name="mi-nj" mode="virtual_bridge" />
   <net name="mi-ny" mode="virtual_bridge" />
   <net name="ga-pa" mode="virtual_bridge" />
   <net name="ny-pa" mode="virtual_bridge" />
   <net name="nj-pa" mode="virtual_bridge" />
   <net name="dc-ny" mode="virtual_bridge" />
```

```
   <net name="dc-nj" mode="virtual_bridge" />
   <net name="dc-tx" mode="virtual_bridge" />
   <net name="ga-tx" mode="virtual_bridge" />


<!--     WA              -->
 <vm name="WA" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="ca1-wa">
      <ipv4>10.0.0.1/24</ipv4>
    </if>
    <if id="2" net="ca2-wa">
      <ipv4>10.0.1.1/24</ipv4>
    </if>
    <if id="3" net="il-wa">
      <ipv4>10.0.18.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     CA1               -->
 <vm name="CA1" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="ca1-wa">
      <ipv4>10.0.0.2/24</ipv4>
    </if>
    <if id="2" net="ca1-ut">
      <ipv4>10.0.3.1/24</ipv4>
    </if>
    <if id="3" net="ca1-ca2">
      <ipv4>10.0.2.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>
<!--     CA2               -->
 <vm name="CA2" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="ca2-wa">
      <ipv4>10.0.1.2/24</ipv4>
    </if>
    <if id="2" net="ca1-ca2">
      <ipv4>10.0.2.2/24</ipv4>
    </if>
```

```
    <if id="3" net="ca2-tx">
      <ipv4>10.0.9.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     UT               -->
 <vm name="UT" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="ca1-ut">
      <ipv4>10.0.3.2/24</ipv4>
    </if>
    <if id="2" net="co-ut">
      <ipv4>10.0.4.1/24</ipv4>
    </if>
    <if id="3" net="mi-ut">
      <ipv4>10.0.5.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>


<!--     CO               -->
 <vm name="CO" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="co-ut">
      <ipv4>10.0.4.2/24</ipv4>
    </if>
    <if id="2" net="co-tx">
      <ipv4>10.0.7.1/24</ipv4>
    </if>
    <if id="3" net="co-ne">
      <ipv4>10.0.6.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     TX               -->
 <vm name="TX" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="ca2-tx">
      <ipv4>10.0.9.2/24</ipv4>
    </if>
```

```
    <if id="2" net="co-tx">
      <ipv4>10.0.7.2/24</ipv4>
    </if>
    <if id="3" net="dc-tx">
      <ipv4>10.0.11.1/24</ipv4>
    </if>
    <if id="4" net="ga-tx">
      <ipv4>10.0.10.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     NE               -->
 <vm name="NE" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="co-ne">
      <ipv4>10.0.6.2/24</ipv4>
    </if>
    <if id="2" net="il-ne">
      <ipv4>10.0.8.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     IL               -->
 <vm name="IL" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="il-ne">
      <ipv4>10.0.8.2/24</ipv4>
    </if>
    <if id="2" net="il-wa">
      <ipv4>10.0.18.2/24</ipv4>
    </if>
    <if id="3" net="il-pa">
      <ipv4>10.0.17.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     PA               -->
 <vm name="PA" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="il-pa">
```

```
      <ipv4>10.0.17.2/24</ipv4>
    </if>
    <if id="2" net="ny-pa">
      <ipv4>10.0.16.1/24</ipv4>
    </if>
    <if id="3" net="nj-pa">
      <ipv4>10.0.19.1/24</ipv4>
    </if>
    <if id="4" net="ga-pa">
      <ipv4>10.0.12.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     MI               -->
 <vm name="MI" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="mi-ut">
      <ipv4>10.0.5.2/24</ipv4>
    </if>
    <if id="2" net="mi-nj">
      <ipv4>10.0.20.1/24</ipv4>
    </if>
    <if id="3" net="mi-ny">
      <ipv4>10.0.15.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     NY              -->
 <vm name="NY" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="mi-ny">
      <ipv4>10.0.15.2/24</ipv4>
    </if>
    <if id="2" net="ny-pa">
      <ipv4>10.0.16.2/24</ipv4>
    </if>
    <if id="3" net="dc-ny">
      <ipv4>10.0.14.1/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     GA               -->
```

```
 <vm name="GA" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="ga-tx">
      <ipv4>10.0.10.2/24</ipv4>
    </if>
    <if id="2" net="ga-pa">
      <ipv4>10.0.12.2/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     DC                -->
 <vm name="DC" type="lxc">
    <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="dc-ny">
      <ipv4>10.0.14.2/24</ipv4>
    </if>
    <if id="2" net="dc-nj">
      <ipv4>10.0.13.1/24</ipv4>
    </if>
    <if id="3" net="dc-tx">
      <ipv4>10.0.11.2/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

<!--     NJ                -->
 <vm name="NJ" type="lxc">
   <filesystem
type="cow">/usr/share/vnx/filesystems/rootfs_lxc_ubuntu</filesyste
m>
    <if id="1" net="dc-nj">
      <ipv4>10.0.13.2/24</ipv4>
    </if>
    <if id="2" net="mi-nj">
      <ipv4>10.0.20.2/24</ipv4>
    </if>
    <if id="3" net="nj-pa">
      <ipv4>10.0.19.2/24</ipv4>
    </if>
  <forwarding type="ip" />
  </vm>

</vnx>
```

Y el script que arrancaba OSPF:

```
<ospf_conf xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="/usr/share/xml/vnx/ospf.xsd">

   <vm name="WA" type="quagga" subtype="lib-install">
      <hostname>vnx</hostname>
      <password>xxxx</password>
      <network area="0.0.0.0">10.0.0.0/16 </network>
      <binaries>
        <zebra>/usr/lib/quagga/zebra</zebra>
        <ospfd>/usr/lib/quagga/ospfd</ospfd>
      </binaries>
   </vm>

   <vm name="CA1" type="quagga" subtype="lib-install">
      <hostname>vnx</hostname>
      <password>xxxx</password>
      <network area="0.0.0.0">10.0.0.0/16</network>
      <binaries>
        <zebra>/usr/lib/quagga/zebra</zebra>
        <ospfd>/usr/lib/quagga/ospfd</ospfd>
      </binaries>
   </vm>

   <vm name="CA2" type="quagga" subtype="lib-install">
      <hostname>vnx</hostname>
      <password>xxxx</password>
      <network area="0.0.0.0">10.0.0.0/16 </network>
      <binaries>
        <zebra>/usr/lib/quagga/zebra</zebra>
        <ospfd>/usr/lib/quagga/ospfd</ospfd>
      </binaries>
   </vm>

   <vm name="UT" type="quagga" subtype="lib-install">
      <hostname>vnx</hostname>
      <password>xxxx</password>
      <network area="0.0.0.0">10.0.0.0/16 </network>
      <binaries>
        <zebra>/usr/lib/quagga/zebra</zebra>
        <ospfd>/usr/lib/quagga/ospfd</ospfd>
      </binaries>
   </vm>

   <vm name="CO" type="quagga" subtype="lib-install">
```

```
      <hostname>vnx</hostname>
      <password>xxxx</password>
      <network area="0.0.0.0">10.0.0.0/16</network>
      <binaries>
        <zebra>/usr/lib/quagga/zebra</zebra>
        <ospfd>/usr/lib/quagga/ospfd</ospfd>
      </binaries>
    </vm>

    <vm name="NE" type="quagga" subtype="lib-install">
      <hostname>vnx</hostname>
      <password>xxxx</password>
      <network area="0.0.0.0">10.0.0.0/16</network>
      <binaries>
        <zebra>/usr/lib/quagga/zebra</zebra>
        <ospfd>/usr/lib/quagga/ospfd</ospfd>
      </binaries>
    </vm>

    <vm name="TX" type="quagga" subtype="lib-install">
      <hostname>vnx</hostname>
      <password>xxxx</password>
      <network area="0.0.0.0">10.0.0.0/16</network>
      <binaries>
        <zebra>/usr/lib/quagga/zebra</zebra>
        <ospfd>/usr/lib/quagga/ospfd</ospfd>
      </binaries>
    </vm>

    <vm name="IL" type="quagga" subtype="lib-install">
      <hostname>vnx</hostname>
      <password>xxxx</password>
      <network area="0.0.0.0">10.0.0.0/16</network>
      <binaries>
        <zebra>/usr/lib/quagga/zebra</zebra>
        <ospfd>/usr/lib/quagga/ospfd</ospfd>
      </binaries>
    </vm>

    <vm name="MI" type="quagga" subtype="lib-install">
      <hostname>vnx</hostname>
      <password>xxxx</password>
      <network area="0.0.0.0">10.0.0.0/16</network>
      <binaries>
        <zebra>/usr/lib/quagga/zebra</zebra>
        <ospfd>/usr/lib/quagga/ospfd</ospfd>
      </binaries>
    </vm>
```

```
<vm name="PA" type="quagga" subtype="lib-install">
   <hostname>vnx</hostname>
   <password>xxxx</password>
   <network area="0.0.0.0">10.0.0.0/16</network>
   <binaries>
     <zebra>/usr/lib/quagga/zebra</zebra>
     <ospfd>/usr/lib/quagga/ospfd</ospfd>
   </binaries>
</vm>

<vm name="GA" type="quagga" subtype="lib-install">
   <hostname>vnx</hostname>
   <password>xxxx</password>
   <network area="0.0.0.0">10.0.0.0/16</network>
   <binaries>
     <zebra>/usr/lib/quagga/zebra</zebra>
     <ospfd>/usr/lib/quagga/ospfd</ospfd>
   </binaries>
</vm>

<vm name="NY" type="quagga" subtype="lib-install">
   <hostname>vnx</hostname>
   <password>xxxx</password>
   <network area="0.0.0.0">10.0.0.0/16</network>
   <binaries>
     <zebra>/usr/lib/quagga/zebra</zebra>
     <ospfd>/usr/lib/quagga/ospfd</ospfd>
   </binaries>
</vm>

<vm name="NJ" type="quagga" subtype="lib-install">
   <hostname>vnx</hostname>
   <password>xxxx</password>
   <network area="0.0.0.0">10.0.0.0/16</network>
   <binaries>
     <zebra>/usr/lib/quagga/zebra</zebra>
     <ospfd>/usr/lib/quagga/ospfd</ospfd>
   </binaries>
</vm>

<vm name="DC" type="quagga" subtype="lib-install">
   <hostname>vnx</hostname>
   <password>xxxx</password>
   <network area="0.0.0.0">10.0.0.0/16</network>
   <binaries>
     <zebra>/usr/lib/quagga/zebra</zebra>
     <ospfd>/usr/lib/quagga/ospfd</ospfd>
   </binaries>
</vm>
```

```
</ospf_conf>
```

Mininet con rutas estáticas:

```
#!/usr/bin/python

"""

"""
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import Node
from mininet.log import setLogLevel, info
from mininet.cli import CLI
import time


from functools import partial

class LinuxRouter( Node ):
    "A Node with IP forwarding enabled."

    def config( self, **params ):
        super( LinuxRouter, self).config( **params )
        # Enable forwarding on the router
        self.cmd( 'sysctl net.ipv4.ip_forward=1' )

    def terminate( self ):
        self.cmd( 'sysctl net.ipv4.ip_forward=0' )
        super( LinuxRouter, self ).terminate()


class NetworkTopo( Topo ):
    "A LinuxRouter connecting three IP subnets"

    def build( self, **_opts ):

        r1 = self.addNode( 'r1', cls=LinuxRouter,
ip='10.0.0.1/16')
     r2 = self.addNode( 'r2', cls=LinuxRouter, ip='10.0.0.2/16')
     r3 = self.addNode( 'r3', cls=LinuxRouter, ip='10.0.2.2/16')
     r4 = self.addNode( 'r4', cls=LinuxRouter, ip='10.0.3.2/16')
     r5 = self.addNode( 'r5', cls=LinuxRouter, ip='10.0.4.2/16')
     r6 = self.addNode( 'r6', cls=LinuxRouter, ip='10.0.11.1/16')
     r7 = self.addNode( 'r7', cls=LinuxRouter, ip='10.0.6.2/16')
     r8 = self.addNode( 'r8', cls=LinuxRouter, ip='10.0.8.2/16')
     r9 = self.addNode( 'r9', cls=LinuxRouter, ip='10.0.5.2/16')
```

```
     r10 = self.addNode( 'r10', cls=LinuxRouter,
ip='10.0.12.1/16')
     r11 = self.addNode( 'r11', cls=LinuxRouter,
ip='10.0.12.2/16')
     r12 = self.addNode( 'r12', cls=LinuxRouter,
ip='10.0.14.1/16')
     r13 = self.addNode( 'r13', cls=LinuxRouter,
ip='10.0.11.2/16')
     r14 = self.addNode( 'r14', cls=LinuxRouter,
ip='10.0.13.2/16')

        self.addLink( r1, r2, intfName1='r1-eth1', intfName2='r2-
eth1',params1= { 'ip' : '10.0.0.1/16' },params2={ 'ip' :
'10.0.0.2/16'})
     self.addLink( r1, r3, intfName1='r1-eth3', intfName2='r3-
eth2', params1= { 'ip' : '10.0.1.1/16' },params2={ 'ip' :
'10.0.1.2/16'})
     self.addLink( r1, r8, intfName1='r1-eth2', intfName2='r8-
eth2', params1= { 'ip' : '10.0.18.1/16' },params2={ 'ip' :
'10.0.18.2/16'})
     self.addLink( r2, r3, intfName1='r2-eth2', intfName2='r3-
eth1', params1= { 'ip' : '10.0.2.1/16' },params2={ 'ip' :
'10.0.2.2/16'})
     self.addLink( r2, r4, intfName1='r2-eth3', intfName2='r4-
eth1', params1= { 'ip' : '10.0.3.1/16' },params2={ 'ip' :
'10.0.3.2/16'})
     self.addLink( r3, r6, intfName1='r3-eth3', intfName2='r6-
eth3', params1= { 'ip' : '10.0.9.1/16' },params2={ 'ip' :
'10.0.9.2/16'})
     self.addLink( r4, r5, intfName1='r4-eth2', intfName2='r5-
eth1', params1= { 'ip' : '10.0.4.1/16' },params2={ 'ip' :
'10.0.4.2/16'})
     self.addLink( r4, r9, intfName1='r4-eth3', intfName2='r9-
eth1', params1= { 'ip' : '10.0.5.1/16' },params2={ 'ip' :
'10.0.5.2/16'})
     self.addLink( r5, r6, intfName1='r5-eth2', intfName2='r6-
eth4', params1= { 'ip' : '10.0.7.1/16' },params2={ 'ip' :
'10.0.7.2/16'})
     self.addLink( r5, r7, intfName1='r5-eth3', intfName2='r7-
eth1', params1= { 'ip' : '10.0.6.1/16' },params2={ 'ip' :
'10.0.6.2/16'})
     self.addLink( r6, r13, intfName1='r6-eth1', intfName2='r13-
eth1',params1= {'ip' : '10.0.11.1/16' },params2={ 'ip' :
'10.0.11.2/16'})
     self.addLink( r6, r11, intfName1='r6-eth2', intfName2='r11-
eth2',params1={ 'ip' : '10.0.10.1/16' },params2={ 'ip' :
'10.0.10.2/16'})
```

```
     self.addLink( r7, r8, intfName1='r7-eth2', intfName2='r8-
eth1', params1= { 'ip' : '10.0.8.1/16' },params2={ 'ip' :
'10.0.8.2/16'})
     self.addLink( r8, r10, intfName1='r8-eth3', intfName2='r10-
eth2',params1={'ip' : '10.0.17.1/16' },params2={ 'ip' :
'10.0.17.2/16'})
     self.addLink( r9, r12, intfName1='r9-eth2', intfName2='r12-
eth2',params1={'ip' : '10.0.15.1/16' },params2={ 'ip' :
'10.0.15.2/16'})
     self.addLink( r9, r14, intfName1='r9-eth3', intfName2='r14-
eth2', params1={'ip' : '10.0.20.1/16' },params2={'ip' :
'10.0.20.2/16'})
     self.addLink( r10, r11, intfName1='r10-eth1', intfName2='r11-
eth1',params1= {'ip' : '10.0.12.1/16' },params2={'ip':
'10.0.12.2/16'})
     self.addLink( r10, r12, intfName1='r10-eth3', intfName2='r12-
eth3',params1={'ip' : '10.0.16.1/16' },params2={'ip' :
'10.0.16.2/16'})
     self.addLink( r10, r14, intfName1='r10-eth4', intfName2='r14-
eth3',params1={'ip' : '10.0.19.1/16' },params2={'ip' :
'10.0.19.2/16'})
     self.addLink( r12, r13, intfName1='r12-eth1', intfName2='r13-
eth3',params1={'ip' : '10.0.14.1/16' },params2={'ip' :
'10.0.14.2/16'})
        self.addLink( r13, r14, intfName1='r13-eth2',
intfName2='r14-eth1',params1={'ip' : '10.0.13.1/16'
},params2={'ip': '10.0.13.2/16'})



def run():
    "Test linux router"
    topo = NetworkTopo()
    privateDirs = [ (
'/home/alejandro/Escritorio/OSPFMININET/quaggacfgs','/etc/quagga'
)]
    r1 = partial( Node,
                  privateDirs=privateDirs )


    net = Mininet( topo = topo )  # controller is used by s1-s3
    net.start()
    print net['r1'].cmd ( './r1config.sh')
    print net[ 'r2' ].cmd( './r2config.sh' )
    print net[ 'r3' ].cmd( './r3config.sh' )
    print net[ 'r4' ].cmd( './r4config.sh' )
    print net[ 'r5' ].cmd( './r5config.sh' )
    print net[ 'r6' ].cmd( './r6config.sh' )
    print net[ 'r7' ].cmd( './r7config.sh' )
```

```
    print net[ 'r8' ].cmd( './r8config.sh' )
    print net[ 'r9' ].cmd( './r9config.sh' )
    print net[ 'r10' ].cmd( './r10config.sh' )
    print net[ 'r11' ].cmd( './r11config.sh' )
    print net[ 'r12' ].cmd( './r12config.sh' )
    print net[ 'r13' ].cmd( './r13config.sh' )
    print net[ 'r14' ].cmd( './r14config.sh' )
    CLI( net )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    run()
```

Donde en cada `rxconfig.sh` se han configurado las rutas correspondientes. A continuación se muestra el ejemplo de una configuración de uno de los nodos:

```
#!/bin/bash
# -*- ENCODING: UTF-8 -*-


echo "Cargando configuracion de red de R1 (WA)"

ifconfig r1-eth1 10.0.0.1 netmask 255.255.0.0 broadcast 10.0.0.255
up
ifconfig r1-eth2 10.0.18.1 netmask 255.255.0.0 broadcast
10.0.18.255 up
ifconfig r1-eth3 10.0.1.1 netmask 255.255.0.0 broadcast 10.0.1.255
up

#/etc/init.d/quagga restart

route add -net 10.0.0.0 netmask 255.255.255.0 gw 10.0.0.1 dev r1-
eth1
route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.1.1 dev r1-
eth3
route add -net 10.0.2.0 netmask 255.255.255.0 gw 10.0.2.1 dev r1-
eth1
route add -net 10.0.3.0 netmask 255.255.255.0 gw 10.0.3.1 dev r1-
eth1
route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.3.1 dev r1-
eth1
route add -net 10.0.5.0 netmask 255.255.255.0 gw 10.0.3.1 dev r1-
eth1
route add -net 10.0.6.0 netmask 255.255.255.0 gw 10.0.8.2 dev r1-
eth2
```

```
route add -net 10.0.7.0 netmask 255.255.255.0 gw 10.0.3.1 dev r1-
eth1
route add -net 10.0.8.0 netmask 255.255.255.0 gw 10.0.8.2 dev r1-
eth2
route add -net 10.0.9.0 netmask 255.255.255.0 gw 10.0.9.1 dev r1-
eth3
route add -net 10.0.10.0 netmask 255.255.255.0 gw 10.0.9.1 dev r1-
eth3
route add -net 10.0.11.0 netmask 255.255.255.0 gw 10.0.9.1 dev r1-
eth3
route add -net 10.0.12.0 netmask 255.255.255.0 gw 10.0.17.1 dev
r1-eth2
route add -net 10.0.13.0 netmask 255.255.255.0 gw 10.0.17.1 dev
r1-eth2
route add -net 10.0.14.0 netmask 255.255.255.0 gw 10.0.17.1 dev
r1-eth2
route add -net 10.0.15.0 netmask 255.255.255.0 gw 10.0.17.1 dev
r1-eth2
route add -net 10.0.16.0 netmask 255.255.255.0 gw 10.0.17.1 dev
r1-eth2
route add -net 10.0.17.0 netmask 255.255.255.0 gw 10.0.17.1 dev
r1-eth2
route add -net 10.0.18.0 netmask 255.255.255.0 gw 10.0.18.1 dev
r1-eth2
route add -net 10.0.19.0 netmask 255.255.255.0 gw 10.0.17.1 dev
r1-eth2
route add -net 10.0.20.0 netmask 255.255.255.0 gw 10.0.17.1 dev
r1-eth2
```

C Figuras de interés

La figura C-1 muestra el consume medio de CPU tanto en frío como en caliente:

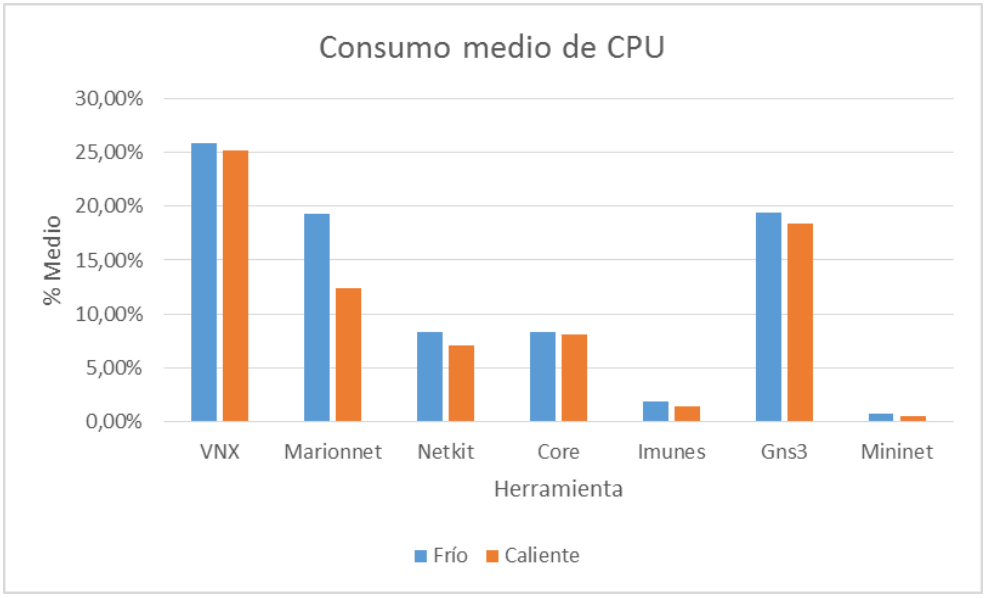

Figura C-2: Consumo medio de CPU

La figura C-2 muestra el consume medio de memoria tanto en frío como en caliente:

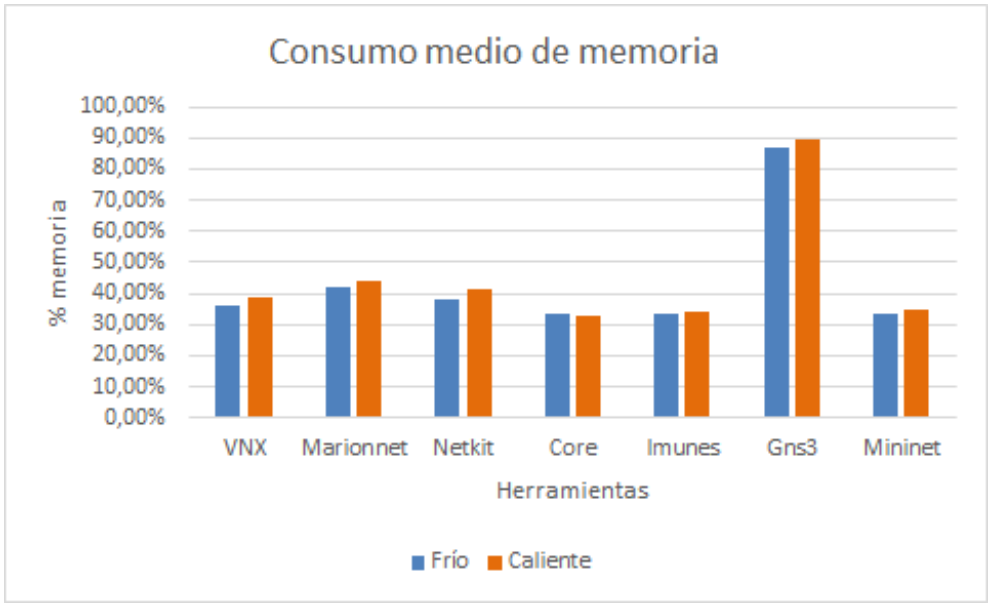

Figura C-3: Consumo medio de memoria